Engin Kirda
Somesh Jha
Davide Balzarotti (Eds.)

LNCS 5758

Recent Advances in Intrusion Detection

12th International Symposium, RAID 2009
Saint-Malo, France, September 2009
Proceedings

Lecture Notes in Computer Science 5758

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Engin Kirda   Somesh Jha
Davide Balzarotti (Eds.)

Recent Advances in Intrusion Detection

12th International Symposium, RAID 2009
Saint-Malo, France, September 23-25, 2009
Proceedings

Springer

Volume Editors

Engin Kirda
Institute Eurecom
2229 Route des Cretes, 06560 Sophia-Antipolis Cedex, France
E-mail: engin.kirda@eurecom.fr

Somesh Jha
University of Wisconsin, Computer Sciences Department
Madison, WI 53706, USA
E-mail: jha@cs.wisc.edu

Davide Balzarotti
Institute Eurecom
2229 Route des Cretes, 06560 Sophia-Antipolis Cedex, France
E-mail: davide.balzarotti@eurecom.fr

Library of Congress Control Number: 2009934013

CR Subject Classification (1998): K.6.5, K.4, E.3, C.2, D.4.6

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN 0302-9743
ISBN-10 3-642-04341-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-04341-3 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12752768 06/3180 5 4 3 2 1 0

Preface

On behalf of the Program Committee, it is our pleasure to present the proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection systems (RAID 2009), which took place in Saint-Malo, France, during September 23–25. As in the past, the symposium brought together leading researchers and practitioners from academia, government, and industry to discuss intrusion detection research and practice. There were six main sessions presenting full research papers on anomaly and specification-based approaches, malware detection and prevention, network and host intrusion detection and prevention, intrusion detection for mobile devices, and high-performance intrusion detection. Furthermore, there was a poster session on emerging research areas and case studies.

The RAID 2009 Program Committee received 59 full paper submissions from all over the world. All submissions were carefully reviewed by independent reviewers on the basis of space, topic, technical assessment, and overall balance. The final selection took place at the Program Committee meeting on May 21 in Oakland, California. In all, 17 papers were selected for presentation and publication in the conference proceedings. As a continued feature, the symposium accepted submissions for poster presentations which have been published as extended abstracts, reporting early-stage research, demonstration of applications, or case studies. Thirty posters were submitted for a numerical review by an independent, three-person sub-committee of the Program Committee based on novelty, description, and evaluation. The sub-committee recommended the acceptance of 16 of these posters for presentation and publication.

The success of RAID 2009 depended on the joint effort of many people. We would like to thank all the authors of submitted papers. We would also like to thank the Program Committee members and additional reviewers, who volunteered their time to evaluate the numerous submissions. In addition, we would like to thank the General Chair, Ludovic Me, for handling the conference arrangements, Davide Balzarotti, for handling the publication, Corrado Leita for publicizing the conference, Christophe Bidan for finding sponsors for the conference, and SUPELEC for hosting the conference website. We would also like to thank our sponsors, DCSSI, INRIA Grand Est, EADS, Alcatel Lucent and Fondation Michel Metivier.

July 2009

Engin Kirda
Somesh Jha

Organization

RAID 2009 was organized by SUPELEC and it was co-located with ESORICS 2009.

Conference Chairs

General Chair	Ludovic Mé (SUPELEC)
Program Chair	Engin Kirda (Eurecom)
Program Co-chair	Somesh Jha (University of Wisconsin)
Sponsorship Chair	Christophe Bidan (SUPELEC)
Publicity Chair	Corrado Leita (Symantec Research Europe)
Publications Chair	Davide Balzarotti (Eurecom)

Steering Committee

Marc Dacier	Symantec Research Europe
Robert Cunnigham	MIT Lincoln Laboratory, USA
Hervé Debar	France Telecom R&D, France
Deborah Frincke	Pacific Northwest National Lab, USA
Ming-Yuh Huang	The Boeing Company, USA
Erland Jonsson	Chalmers University, Sweden
Christopher Kruegel	University of California, Santa Barbara, USA
Wenke Lee	Georgia Tech, USA
Richard Lippmann	MIT Lincoln Laboratory
Ludovic Mé	SUPELEC, France
Alfonso Valdes	SRI International, USA
Giovanni Vigna	University of California, Santa Barbara, USA
Andreas Wespi	IBM Research, Switzerland
S. Felix Wu	UC Davis, USA
Diego Zamboni	IBM Research, Switzerland

Program Committee

Anil Somayaji	Carleton University, Canada
Benjamin Morin	Central Directorate for Information System Security, France
Christopher Kruegel	University of California, Santa Barbara, USA
Collin Jackson	Stanford University, USA
Corrado Leita	Symantec Research Europe, France
David Brumley	Carnegie Mellon University, USA
Davide Balzarotti	Eurecom, France

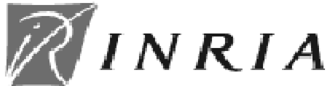
Dongyan Xu	Purdue University, USA
Engin Kirda	Eurecom, France
Giovanni Vigna	University of California, Santa Barbara, USA
Guevara Noubir	North Eastern University, USA
Guofei Gu	Texas A&M University, USA
Jaeyeon Jung	Intel Research, USA
John Viega	Stonewall Software, USA
Jonathan Giffin	Georgia Institute of Technology, USA
Jouni Viinikka	Orange Labs, France
Kathy Wang	MITRE, USA
Manuel Costa	Microsoft Research, Cambridge, UK
Michael Bailey	University of Michigan, USA
Mihai Christodorescu	IBM T.J. Watson Research Center, USA
Olivier Festor	INRIA, France
R. Sekar	Stoney Brook University, USA
Radu State	University of Luxembourg, Luxembourg
Robert Cunningham	MIT Lincoln Labs, USA
Robin Sommer	International Computer Science Institute, USA
Somesh Jha	University of Wisconsin, USA
Sotiris Ioannidis	FORTH, Greece
Thorsten Holz	University of Mannheim, Germany
Xuxian Jiang	North Carolina State University, USA

Additional Reviewers

Rémi Badonnel	INRIA, France
Adam Barth	UC Berkeley, USA
Drew Davidson	University of Wisconsin, USA
Matt Fredrickson	University of Wisconsin, USA
Zhiqiang Lin	Purdue University, USA
Daniel Luchaup	University of Wisconsin, USA
Roberto Perdisci	Damballa, Inc., USA
Ryan Riley	Purdue University, USA
Elizabeth Stinson	Stanford University, USA
Alok Tongoankar	Stony Brook University, USA
Zhi Wang	North Carolina State University, USA

Sponsoring Institutions

DCSSI (Direction centrale de la sécurité des systèmes d'information)



INRIA Grand Est

EADS

Alcatel Lucent

Fondation Michel Metivier

Table of Contents

Recent Advances in Intrusion Detection Anomaly and Specification-Based Approaches

Panacea: Automating Attack Classification for Anomaly-Based Network Intrusion Detection Systems 1
Damiano Bolzoni, Sandro Etalle, and Pieter H. Hartel

Protecting a Moving Target: Addressing Web Application Concept Drift 21
Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna

Adaptive Anomaly Detection via Self-calibration and Dynamic Updating 41
Gabriela F. Cretu-Ciocarlie, Angelos Stavrou, Michael E. Locasto, and Salvatore J. Stolfo

Runtime Monitoring and Dynamic Reconfiguration for Intrusion Detection Systems 61
Martin Rehák, Eugen Staab, Volker Fusenig, Michal Pěchouček, Martin Grill, Jan Stiborek, Karel Bartoš, and Thomas Engel

Malware Detection and Prevention (I)

Malware Behavioral Detection by Attribute-Automata Using Abstraction from Platform and Language 81
Grégoire Jacob, Hervé Debar, and Eric Filiol

Automatic Generation of String Signatures for Malware Detection 101
Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh

PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime 121
M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq

Network and Host Intrusion Detection and Prevention

Automatically Adapting a Trained Anomaly Detector to Software Patches 142
Peng Li, Debin Gao, and Michael K. Reiter

Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration 161
Juan Caballero, Zhenkai Liang, Pongsin Poosankam, and Dawn Song

Automated Behavioral Fingerprinting ... 182
Jérôme François, Humberto Abdelnur, Radu State, and Olivier Festor

Intrusion Detection for Mobile Devices

SMS-Watchdog: Profiling Social Behaviors of SMS Users for Anomaly Detection ... 202
Guanhua Yan, Stephan Eidenbenz, and Emanuele Galli

Keystroke-Based User Identification on Smart Phones ... 224
Saira Zahid, Muhammad Shahzad, Syed Ali Khayam, and Muddassar Farooq

VirusMeter: Preventing Your Cellphone from Spies ... 244
Lei Liu, Guanhua Yan, Xinwen Zhang, and Songqing Chen

High-Performance Intrusion Detection

Regular Expression Matching on Graphics Hardware for Intrusion Detection ... 265
Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis

Multi-byte Regular Expression Matching with Speculation ... 284
Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha

Malware Detection and Prevention (II)

Toward Revealing Kernel Malware Behavior in Virtual Execution Environments ... 304
Chaoting Xuan, John Copeland, and Raheem Beyah

Exploiting Temporal Persistence to Detect Covert Botnet Channels ... 326
Frederic Giroire, Jaideep Chandrashekar, Nina Taft, Eve Schooler, and Dina Papagiannaki

Posters

An Experimental Study on Instance Selection Schemes for Efficient Network Anomaly Detection ... 346
Yang Li, Li Guo, Bin-Xing Fang, Xiang-Tao Liu, and Lin-Qi

Automatic Software Instrumentation for the Detection of Non-control-data Attacks ... 348
Jonathan-Christofer Demay, Éric Totel, and Frédéric Tronel

BLADE: Slashing the Invisible Channel of Drive-by Download Malware 350
Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee

CERN Investigation of Network Behaviour and Anomaly Detection 353
Milosz Marian Hulboj and Ryszard Erazm Jurga

Blare Tools: A Policy-Based Intrusion Detection System Automatically Set by the Security Policy 355
Laurent George, Valérie Viet Triem Tong, and Ludovic Mé

Detection, Alert and Response to Malicious Behavior in Mobile Devices: Knowledge-Based Approach 357
Asaf Shabtai, Uri Kanonov, and Yuval Elovici

Autonomic Intrusion Detection System 359
Wei Wang, Thomas Guyet, and Svein J. Knapskog

ALICE@home: Distributed Framework for Detecting Malicious Sites.... 362
Ikpeme Erete, Vinod Yegneswaran, and Phillip Porras

Packet Space Analysis of Intrusion Detection Signatures 365
Frédéric Massicotte

Traffic Behaviour Characterization Using NetMate 367
Annie De Montigny-Leboeuf, Mathieu Couture, and Frederic Massicotte

On the Inefficient Use of Entropy for Anomaly Detection........ 369
Mobin Javed, Ayesha Binte Ashfaq, M. Zubair Shafiq, and Syed Ali Khayam

Browser-Based Intrusion Prevention System 371
Ikpeme Erete

Using Formal Grammar and Genetic Operators to Evolve Malware 374
Sadia Noreen, Shafaq Murtaza, M. Zubair Shafiq, and Muddassar Farooq

Method for Detecting Unknown Malicious Executables........ 376
Boris Rozenberg, Ehud Gudes, Yuval Elovici, and Yuval Fledel

Brave New World: Pervasive Insecurity of Embedded Network Devices 378
Ang Cui, Yingbo Song, Pratap V. Prabhu, and Salvatore J. Stolfo

DAEDALUS: Novel Application of Large-Scale Darknet Monitoring for Practical Protection of Live Networks (Extended Abstract)........ 381
Daisuke Inoue, Mio Suzuki, Masashi Eto, Katsunari Yoshioka, and Koji Nakao

Author Index........ 383

Panacea: Automating Attack Classification for Anomaly-Based Network Intrusion Detection Systems*

Damiano Bolzoni[1], Sandro Etalle[1,2], and Pieter H. Hartel[1]

[1] University of Twente, Enschede, The Netherlands
[2] Eindhoven Technical University, The Netherlands
{damiano.bolzoni,pieter.hartel}@utwente.nl, s.etalle@tue.nl

Abstract. Anomaly-based intrusion detection systems are usually criticized because they lack a classification of attacks, thus security teams have to manually inspect any raised alert to classify it. We present a new approach, Panacea, to automatically and systematically classify attacks detected by an anomaly-based network intrusion detection system.

Keywords: attack classification, anomaly-based intrusion detection systems.

One of the often cited weaknesses of anomaly-based intrusion detection systems (ABSs) is the fact that they cannot classify the attacks they detect (Ghosh and Schwartzbard [1] and Robertson et al. [2]). The lack of an attack classification affects the overall usability of an ABS, because security teams have to manually process each alert the ABS raises in order to assess the impact of the detected attack, and to handle the alert.

Today, security teams faces two main challenges. First, because the most harmful attacks currently consist of several stages (Ning et al. [3]), security teams need to detect an attack at the earliest stage, in order to stop it. Secondly, because of the activities conducted by automatic scanners, BOTnets, and script-kiddies the number of security alerts has increased over the years. Although true positives when detected by an IDS, these kinds of activities cannot normally be considered a serious threat, i.e., they are "non-relevant" events (e.g., a remote automatic scanner attempting to replicate a 5-year old attack against a now-secure PHP script).

A number of automatic techniques to perform alert correlation have been proposed (Cuppens and Ortalo [4], Debar and Wespi [5], Ning and Xu [6] and Valeur et al. [7]), in order to detect attacks at an early stage, or lower the false and the non-relevant alert rates. However, such techniques require a good deal of

* This research is supported by the research program Sentinels (http://www.sentinels.nl). Sentinels is being financed by Technology Foundation STW, the Netherlands Organization for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 1–20, 2009.
© Springer-Verlag Berlin Heidelberg 2009

information (apart from the usual IP addresses and TCP ports) to be effective: the attacks that triggered the alerts must be classified.

By classifying an attack (e.g., buffer overflow, SQL Injection), it is also possible to set default actions for handling a certain alert. The alert could (1) trigger automatic countermeasures, e.g., either because an early attack stage has been detected or because the attack class is considered to have a great impact on the security. Alternately, the alert could be (2) forwarded for manual handling or (3) filtered and stored for later analysis (i.e., correlation) and statistics.

Determining the class of an attack is trivial for an alert generated by a signature-based IDS (SBS), like Snort [8,9]. Each signature is the result of an analysis of the corresponding attack conducted by experts: the attack class is manually assigned during the signature development process (i.e., the alert class is included in the signature). Thus, security teams usually do not need to further process the alert to assign a class, and they can set precisely a standard action for the system to execute when such an alert is triggered.

Problem. When an ABS raises an alert, it cannot associate the alert with an attack class. The system detects an anomaly, but it has too little information (typically only source and destination IP addresses and TCP ports) to determine the attack class. No automatic or semi-automatic approach is currently available to classify anomaly-based alerts. Thus, any anomaly-based alert must be manually processed to identify the alert class, increasing the workload of security teams. A solution to automate the classification of anomaly-based alerts is to employ some heuristics (e.g., see Robertson et al. [2]) to analyse the anomaly-based alert for features of well-known attacks. Although this approach could lead to good results, it totally relies on the manual implementation of the heuristics (which could be a labour intensive task), and on the expertise of the operator.

Contribution. In this paper we present Panacea, a simple, yet effective, system that uses machine learning techniques to automatically and systematically classify attacks detected by a payload-based ABS (and consequently the generated alerts as well). The basic idea is the following. Attacks that share some common traits, i.e., some byte sequences in their payloads, are usually in the same class. Thus, by extracting byte sequences from an alert payload (triggered by a certain attack), we can compare those sequences to previously collected data with an appropriate algorithm, find the most similar alert payload, and then infer the attack class from the matching alert payload class.

To the best of our knowledge, Panacea is the first system proposed that:

– Automatically classifies attacks detected by an ABS, without using predetermined heuristics;
– Does not need manual assistance to classify attacks (with some exceptions to be described in Section 1.1).

Panacea requires a training phase for its engine to build the attack classifier. Once the training phase is complete, Panacea classifies any attack detected by the ABS automatically. Here we consider only attacks that target networks, however it is possible to extend the approach to include host-based IDSs too.

Limitation of the approach. Panacea analyses the generated alert payload to build its classification model. Thus, any alert generated by attacks/activities that do not involve a payload (e.g., a port scan or a DDoS) cannot be automatically classified. As most of the harmful attacks inject some data in target systems, we do not see this as a serious limitation.

This paper is organized as follows. In Section 1 we present the architecture of Panacea, we detail its components, the way they interact and the data they exchange (Section 1.1). In Section 1.2, we summarise the machine learning algorithms that Panacea uses to classify the alerts. In Section 2 we show the results of the benchmarks. Section 3 presents related work, while Section 4 concludes.

1 Architecture

Panacea consists of two interacting components: the Alert Information Extractor (AIE) and the Attack Classification Engine (ACE). The AIE receives alerts from the IDS(s), processes the payload, and extracts significant information, outputting alert meta-information. This meta-information is then passed to the ACE that automatically determines the attack class. The classification process goes through two main stages. First, the ACE is trained with several types of alert meta-information to build a classification model. The ACE is fed alert meta-information and the corresponding attack class. The attack class information can be provided in several ways, either manually by an operator or automatically by extracting additional information from the original alert (only when the alert has been raised by an SBS). Secondly, when the training is completed, the ACE is ready to classify new incoming alerts automatically. We now describe each component and the working modes of Panacea in detail. Figure 1 depicts Panacea and its internal components.

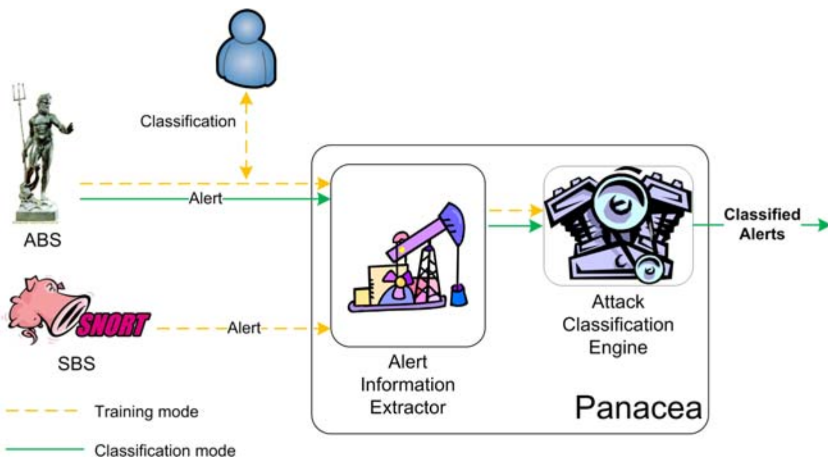


Fig. 1. An overview of the Panacea architecture and the internal components

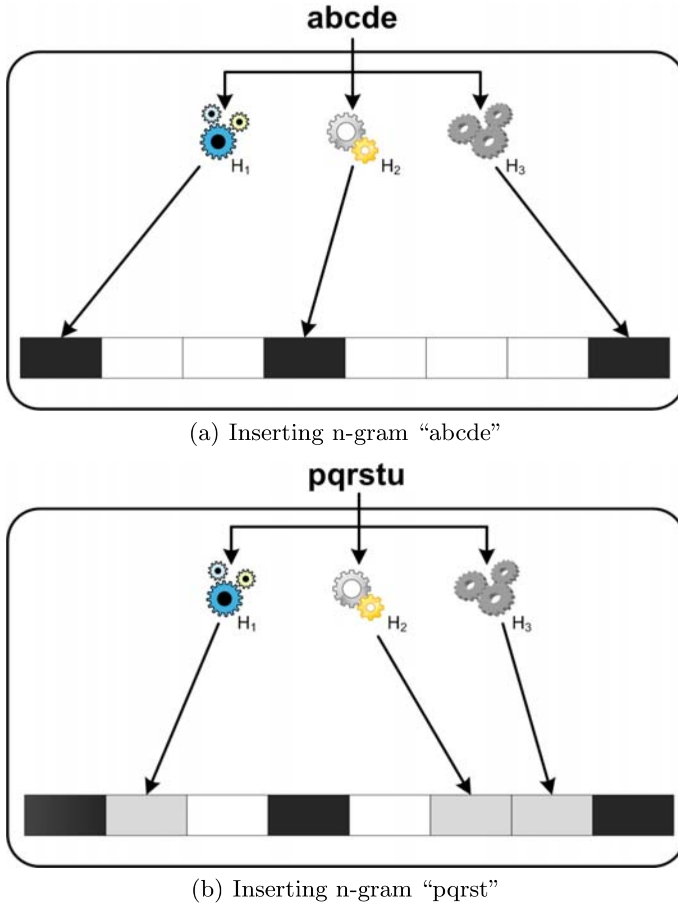
1.1 Alert Information Extractor

The first component we examine is the AIE. The extraction of relevant information from alert payloads is a crucial step, as it is the basis for attack class inference. Requirements for this phase are that the extraction function should capture enough features from the original information (i.e., the payload) to distinguish alerts belonging to different classes, and it should be efficient w.r.t. the required memory space. We now describe the analysis techniques we have chosen.

Extracting and storing relevant information. N-gram analysis [10] allows one to capture features of data in an efficient way, and it has been used before in the context of computer security to detect attacks (Forrester and Hofmeyr [11], Wang and Stolfo [12]). N-gram analysis is a suitable technique to capture data features also for the problem of attack classification, and the AIE employs such a technique to extract relevant information from alert payloads.

As Wang et al. note [13], by using higher order n-grams (i.e., n-grams where $n > 1$) it is possible to capture more data features and to achieve a more precise analysis. One has to consider that the whole feature space size of a higher-order n-gram is 256^n (where n is the n-gram order). The comparison of byte frequency values becomes quickly infeasible, also for values of n such as 3 or 4, because the space needed to store average and standard deviation values for each n-gram grows exponentially (e.g., 640GB would be needed to store 5-grams statistics). Although a frequency-based n-gram analysis may seem to model data distribution accurately, Wang et al. experimentally show that a binary-based n-gram analysis is more precise in the context of network data analysis. In practice, the fact that a certain n-gram has occurred is stored, rather than computing average byte frequency and standard deviation statistics. The reason why the binary approach performs better is that high-order n-grams are more sparse than low-order n-grams, thus it is more difficult to gather accurate byte-frequency statistics as the order increases. This approach has an additional advantage, other than being more precise. Because less information is required, it requires less space in memory, and we can consider higher-order n-grams (such as 5). We now present the data structure used by the AIE to store the extracted information.

Bloom filter. A Bloom filter [14] (BF) is a method to represent a set of S elements (n-grams in our embodiment) in a smaller space. Formally, a BF is a pair $\langle b, H \rangle$ where b is a bit map of l bits, initially all set to 0, and H is a set of k independent hash functions $h_1 \dots h_k$. H determines the storage of b in such a way that, given an element s in S: $\forall h_k,\ b_i = 1 \iff h_k(s)\ mod\ l = i$. In other words, for each n-gram s in S, and for each hash function h_k, we compute $h_k(s)\ mod\ l$, and we use the resulting value as index to set to 1 the bit in b corresponding to it. When checking for the presence of a certain element s, the element is considered to be stored in the BF if: $\forall h_k,\ b_{h_k(s)\ mod\ l} = 1$. Because of the n-gram sparsity, a BF with a size of 10KB is sufficiently large to store the alert meta-information resulting from 5-grams analysis.



(a) Inserting n-gram "abcde"

(b) Inserting n-gram "pqrst"

Fig. 2. Examples of inserting two different 5-grams. H_1, H_2 and H_3 represent different hash functions.

A BF employs k different hash functions at the same time to decrease the probability of a false positive (the opposite situation, a false negative, cannot occur). False positives occur when all of the bit positions calculated for a given element have been set to 1 when inserting previous elements, due to the collisions generated by hash functions. The false positive rate for a given BF is $(1 - e^{\frac{kn}{l}})^k$, where n is the number of elements already stored.

Operational modes. The AIE not only extracts information from alerts as described above, but it is also responsible for forwarding the attack class information to the classification engine, when the latter is in training mode. The attack class can be provided either automatically or manually. In case an SBS is deployed next to the ABS and it is monitoring the same data, it is possible to

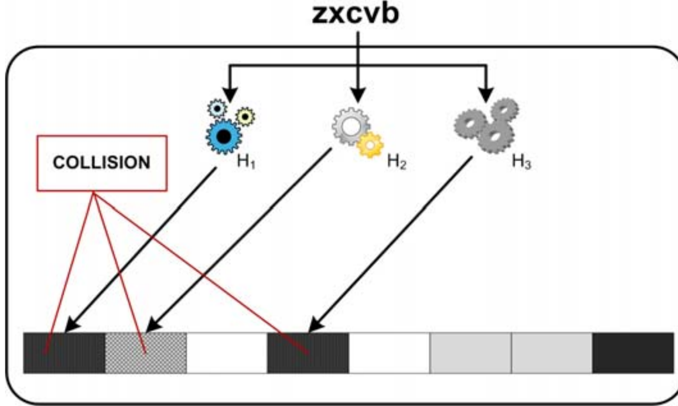


Fig. 3. Example of a false positive. The element “zxcvb” has not been inserted in the Bloom filter. Due to the collisions generated by the hash functions, the test for its presence returns “true”.

feed the ACE during training both the payload and the attack class of any alert generated by the SBS. We define this as the *automatic* mode, since no human operator is required to carry out the attack classification. A human operator can classify the alerts raised by the ABS (in a manner consistent with the SBS classification), hence integrating those with the alerts raised by the ABS during the ACE training. We call this the *semi-automatic* mode. The last possible operative mode is the *manual* mode. In this case, *any* alert is manually classified by an operator.

Each mode presents advantages and disadvantages. In automatic mode, the workload is low, but on the other hand the classification accuracy is likely to be low as well. In fact, the SBS and the ABS are likely to detect different attacks, hence the classification engine could be trained to correctly classify only a subset of the ABS alerts. The manual mode requires human intervention but it is likely to produce better results, since each alert is consistently classified. We assume that the alerts raised by the SBS and ABS have already been verified and any false positive alert has already been purged (e.g., using ALAC [15] or our ATLANTIDES [16]).

1.2 Attack Classification Engine

The ACE includes the algorithm used to classify attacks. Since we are aware of the attack class information, we consider only *supervised* machine learning algorithms. These algorithms generally achieve better results than unsupervised algorithms (where the algorithm, e.g. K-medoids, deduces classes by measuring inter-data similarity). The classification algorithm must meet several requirements, namely:

- support for multiple classes, as alerts fall in several classes;
- classification of high-dimensional data, since each bit of the BF data structure the ACE receives in input is seen as a dimension of analysis;
- fast training (the reason for this will be clarified later) and classification phases;
- (optional) estimate classification confidence when in classification phase.

We consider the last requirement optional, as it does not directly influence the quality of the classification, though it is useful to improve the system usability. Confidence measures the likelihood of having a correct classification for a given input. Users can instruct the system to forward any alert whose confidence value is lower than a given threshold for manual classification, hence reducing the probability of misclassification (at the price of an increased workload, see Section 2.7).

We chose two algorithms for our experiments: (1) Support Vector Machines (SVM) and (2) the RIPPER rule learner. These algorithms implement supervised techniques, their training and classification phases are fast and handle high-dimensional data. Both algorithms perform non-incremental learning. A non-incremental algorithm iterates on samples several times to build the best classification model by minimizing the classification error. The whole training set is then needed at once, and additional samples cannot be incorporated in the classification model unless the training phase is run from scratch. On the other hand, an incremental algorithm can modify the model after the main training phase as new samples become available. An incremental algorithm usually performs worse than a non-incremental algorithm, because the model is not re-built. Thus, a non-incremental algorithm is the best choice to perform an accurate classification. However, because it is highly unlikely that we can collect all alerts for training at once the choice of non-incremental algorithms could be seen as a limitation of our system.

In practice, thanks to the limited BF memory size, we can store a huge number of samples and, by applying a "batch training", we can simulate incremental learning in non-incremental algorithms. As new training samples become available, we add them to the batch training set and build the classifier using the entire set only when a certain number of samples is reached. Then, the classifier is re-built with the set of "batches" available at that time. Because both SVM and RIPPER are fast in training, there are no computational issues.

We chose SVM and RIPPER, not only because they meet the requirements, but for two additional reasons. First, they yield high-quality classifications. Meyer et al. [17] test the SVM against several other classification algorithms (available from the R project [18]) on real and synthetic data sets. An SVM outperforms competitors in 50% of tests and ranks in the top 3 in 90% of them. RIPPER has been used before in the context of intrusion detection (e.g., on data relative to system calls and network connections [19,20]) with good results. Secondly, because they approach the classification problem differently (geometric for SVM, and rule-based for RIPPER), the algorithms are supposed to compensate for each others weaknesses. Hence, we can evaluate which algorithm is more suitable in different contexts. We will now provide some detail on the algorithms.

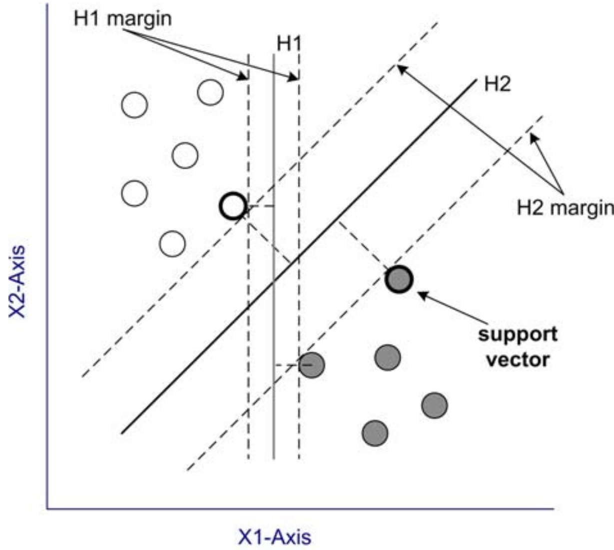


Fig. 4. Hyperplanes in a 2-dimensional space. H1 separates samples sets with a small margin, H2 does that with the maximum margin. The example refers to linearly separable data. The support vectors are shown with a thicker border.

Support Vector Machines. (Vapnik and Lerner [21]) is a set of supervised learning methods used for classification. In the original formulation, an SVM is a binary classifier. It uses a non-linear mapping to transform the original training data into a higher dimension. Then, it searches for the linear optimal separating hyperplane, i.e., a plane that separates the samples of one class from another. An SVM uses "support vectors" and "margins" to find the optimal hyperplane, i.e., the plane with the maximum margin.

The original SVM algorithm has been modified to classify non-linear data and to use multiple classes. Boser et al. [22] introduce non-linear data classification by using kernel functions (i.e., non-linear functions). To support multiple classes, the problem is reduced to multiple binary sub-problems. Given m classes, m classifiers are trained, one for each class. Any test sample is assigned to the class corresponding to the largest positive distance.

RIPPER. (Cohen [23]) is a fast and effective rule induction algorithm. RIPPER uses a set of IF-THEN rules. An IF-THEN rule is an expression in the form **IF** <*condition*> **THEN** <*conclusion*>. The IF-part of a rule is called the rule antecedent. The THEN-part is the rule consequent. The *condition* consists of one or more attribute tests, that are logically ANDed. A test t_i is in the form $t_i = v$ for categorical attributes (where v is a categorical label) or either $t_i \geq \theta$ or $t_i \leq \theta$ for numerical attributes (where θ is a numerical value). The conclusion contains a class prediction. If, for a given input, the condition (i.e., all of the attribute

tests) holds true, then the rule antecedent is satisfied and the corresponding class in the conclusion is returned (the rule is said to "cover" the input). Since RIPPER employs ordered rules, when a match occurs, the algorithm does not evaluate other rules. Some examples of rules are:

IF $bf[i] = 1$ *AND* ... *AND* $bf[k] = 1$ **THEN** *class = cross-site scripting*
IF $bf[l] = 1$ *AND* ... *AND* $bf[n] = 1$ **THEN** *class = sql injection*

RIPPER builds the rule set for a certain class SC_i as follows. The training data set is split into two sets, a pruning and a growing sets. The classifier is built using these two sets by repeatedly inserting rules starting from an empty rule set (the growing set). The algorithm heuristically adds one condition at a time until the rule has no error rate on the growing set. RIPPER implements also an optimisation phase, in order to simplify the rule set.

When multiple classes $C_1 \ldots C_n$ are used, RIPPER sorts classes on a sample frequency basis and induces rules sequentially from the least prevalent class SC_1 to the second most prevalent class SC_{n-1}. The most prevalent class SC_n becomes the *default* class, and no rule is induced for it (thus, in case of a binary classification, RIPPER induces rules for the minority class only).

1.3 Implementation

We have implemented a prototype of Panacea to run our experiments. The prototype is written in Java, since we link to the libraries provided by the Weka platform [24]. Weka is a well-known collection of machine learning algorithms, and it contains an implementation of both SVM and RIPPER. Weka provides also a comprehensive framework to run benchmarks on several data sets under the same testing conditions. The attacks samples generated by network IDSs, in the form of alerts, are stored in a database that the system fetches to extract the alert payload information.

2 Benchmarks

Public data sets for benchmarking IDSs are scarce. It is even more difficult to find a suitable data set to test Panacea, since no research has systematically addressed the problem of (semi)automatically classifying attacks detected by an ABS before. Hence, we have collected three different data sets (referred to as DS_A, DS_B and DS_C, see below for a description of the data sets) to evaluate the accuracy of Panacea. These data sets are used to evaluate the accuracy of Panacea in different scenarios: (1) when working in automatic mode (DS_A), (2) when using an ad hoc taxonomy and the manual mode (DS_B) and (3) when classifying unknown attacks (e.g., generated by two ABSs), having trained the system with alerts from known attacks (DS_B and DS_C).

In the literature there are several taxonomies and classifications of security events. For instance, Howard [25], Hansman and Hunt [26], and the well-known taxonomy used in the DARPA 1998 [27] and 1999 [28] data sets. Only the latter

classification has been used in practice (in spite of its course granularity, as it contains only four classes which are unsuitable to classify modern attacks). In our experiments, we use the Snort classification for benchmarks with DS_A (see [29] for a detailed taxonomy) and the Web Application Security Consortium Threat Classification [30] for benchmarks with DS_B and DS_C.

To evaluate the accuracy of the classification model, we use two approaches. For test (1) and (2), we employ cross-validation. In cross-validation, samples are partitioned into sub-sets. The analysis is first performed on a single sub-set, while the other sub-set(s) are retained to validate the initial analysis. In k-fold cross-validation, the samples are partitioned into k sub-sets. A single sub-set is retained as the validation data for testing the model, and the remaining $k-1$ sub-sets are used as training data to build the model. The process is repeated k times (the "folds"), using each of the k sets exactly once to validate the model. Usually the k fold results are combined (e.g., averaged) to generate a single estimation. The advantage of this method is that all of the samples are used for both training and validation, and each sample is used for validation exactly once. We use 10 folds in our experiments, which is a standard value, used in the Weka testing environment too.

For test 3), we use one of DS_B and DS_C for training and the other for testing. The accuracy is evaluated by counting the number of correctly classified attacks.

2.1 Data Sets

DS_A. contains alerts raised by Snort (see Table 1 for alert figures). To collect the largest number of alerts possible, we have used several tools to automatically inject attack payloads (Nessus [31] and a proprietary vulnerability assessment tool). Attacks have been directed against a system running some virtual machines with both Linux- and Windows-based installations, which expose several services (e.g., web server, DBMS, web proxy, SMTP and SSH). We collected more than 3200 alerts in total, classified in 14 different (Snort) attack classes. However, some classes have few alerts, thus we select only classes with at least 10 alerts. This data set (and DS_B as well) is synthetic. We do not see this as a limitation since the alerts cover multiple classes and trigger a large number of different signatures. We test how the system behaves in automatic mode, the whole set being generated by Snort.

DS_B. contains a set of more than 1400 Snort alerts related to web attacks (Table 2 provides alert details). To generate this data set, we have used Nessus, Nikto [32] (a web vulnerability scanner), and we have manually injected attack payloads collected from the well-known site Milw0rm, that hosts a large collection of web exploits [33]. The attack classification has been performed manually (manual mode), since Snort does not provide a fine-grained classification of web-related attacks (alerts are allocated to different classes with other alerts, see Table 1). Attacks have been classified according to the Web Application Security Consortium Threat Classification [30].

Table 1. DS_A (alerts raised by Snort): attack classes and samples. It is not surprising that web-related attacks account for more than 50%, since most Snort signatures address web vulnerabilities. * marks classes that contain web-related attacks.

Attack Class	Description	# of samples
attempted-recon*	Attempted information leak	1379
web-application-attack*	Web application attack	1032
web-application-activity*	Access to a potentially vulnerable web application	599
unknown	Unknown traffic	66
attempted-user*	Attempted user privilege gain	45
misc-attack	Miscellaneous attack	44
attempted-admin	Attempted administrator privilege gain	32
attempted-dos	Attempted Denial of Service	14
bad-unknown	Potentially bad traffic	13

Table 2. DS_B: attack classes and samples. Attacks have been classified according to the Web Application Security Consortium Threat Classification.

Attack Class	# samples
Path Traversal	931
Cross-site Scripting	399
SQL Injection	73
Buffer Overflow	8

Table 3. DS_C: attack classes and samples. Attacks have been classified according to the Web Application Security Consortium Threat Classification.

Attack Class	# samples
Path Traversal	53
Cross-site Scripting	27
SQL Injection	16
Buffer Overflow	4

DS_C is a collection of alerts generated over a period of 2 weeks by two ABSs, i.e., our POSEIDON [34] and Sphinx [35]. POSEIDON is a general-purpose anomaly-network-based IDS, which uses a combination of a neural network with the well-know algorithm PAYL [12] to analyse network data and detect attacks. Sphinx is a web- anomaly-based IDS, which analyses HTTP request parameters and which detects data-flow [36] attacks. We recorded network traffic directed to a main web server of the university network, and did not inject any attack. Afterwards, we processed this data with POSEIDON and Sphinx to generate alerts. The inspection of alerts and the classification of attacks has been performed manually (using the same taxonomy we apply for DS_B). The data set consists of a set of 100 alerts, and Table 3 reports attack details.

2.2 Tests with DS_A

We use DS_A to validate the general effectiveness of our approach. There are three factors which influence the classification accuracy, namely: (1) the number of alerts processed during training, (2) the length of n-grams used, and (3) the classification algorithm selected. This preliminary test aims to identify which parameter combination(s) results in the most accurate classification.

Testing methodology. We proceed with a 3-step approach. First, we want to identify an adequate number of samples required for training: in fact, a too low number of samples could generate an inaccurate classification. On the other hand, while it is generally a good idea to have as many training samples as possible, after some point the benefit from adding additional information could become negligible. Secondly, we want to identify the best n-gram length. Short n-grams are likely to be shared among many attack payloads, and the attack diversification would be poor (i.e., a number of different attacks contains the same n-grams). On the other hand, long n-grams are unlikely to be common among attack payloads, hence it would be difficult to predict a class for a new attack that does not share a sufficient number of long n-grams. Finally, we analyse how the classification algorithms work by analysing the overall classification accuracy (i.e., considering all of the attack classes) and the per-class accuracy. The two algorithms approach the classification problem in two totally different ways, and each of them could be performing better under different circumstances.

To avoid bias by choosing a specific attack, we randomly select alerts in the sub-sets. In fact, by selecting alerts for training in the same order they have been generated (as opposed to random), we could end up with few (or no) samples in certain classes, hence influencing the accuracy rate (i.e., a too good, or bad, value). To enforce randomness, we also run several trials (five) with different sub-sets and calculate the average accuracy rate. Table 4 reports benchmark results (the percentage of correctly classified attacks) for SVM and RIPPER.

Discussion. Tests with DS_A indicate that the approach is effective in classifying attacks. As the number of training samples increases, accuracy increases as well for both algorithms. Also the n-gram length directly influences the classification.

Table 4. Test results on DS_A with SVM and RIPPER. We report the average percentage of correctly classified attacks of five trials. As the number of samples in the testing sub-set increases, the overall effectiveness increases as well. Longer n-grams generally produce better results, up to length 3. SVM performs better than RIPPER by a narrow margin.

	SVM n-gram length				**RIPPER n-gram length**			
# samples	1	2	3	4	1	2	3	4
1000	62.6%	76.8%	**77.3%**	76.7%	66.1%	75.9%	**76.2%**	75.7%
2000	65.9%	78.6%	**78.9%**	77.7%	69.4%	76.7%	**76.9%**	76.4%
3000	66.3%	79.4%	**79.6%**	78.6%	72.7%	77.2%	**77.5%**	76.9%

Table 5. Per-class detailed results on DS_A, using 3-grams. We report the average percentage of correctly classified attacks of five trials. RIPPER performs better than SVM in classifying all attacks, .

	SVM # of samples			**RIPPER** # of samples		
Attack Class	1000	2000	3000	1000	2000	3000
attempted-recon	**90.9%**	90.5%	90.7%	90.4%	93.9%	**94.0%**
web-application-attack	79.8%	**89.0%**	88.8%	97.4%	98.8%	**99.1%**
web-application-activity	80.8%	**81.2%**	80.9%	93.7%	**96.1%**	95.8%

The number of correctly classified attacks increases as n-grams get longer, up to 3-grams. N-grams of length 4 produce a slightly worse classification, and the same happens for 1-grams (which achieve the worst percentages). SVM and RIPPER present similar accuracy rates on 3-grams, with the former being slightly better. However, if we perform an analysis based on per-class accuracy (see Table 5), we observe that, although both classification algorithms score high on accuracy level for the three most populated classes, RIPPER is far more precise than SVM (in once case, the "web-application-activity" class, by nearly 15%).

When we look at the overall accuracy rate, averaged among the 9 classes, for DS_A, SVM performs better because of the classes with few alerts. If we zoom into the classes with a significant number of samples, we observe an opposite behaviour. This means that, with a high number of samples, RIPPER performs better than SVM.

In Table 5, a sub-set with fewer samples seems to achieve better results (although percentages differ by a narrow margin), when considering the same algorithm. This happens for SVM once when using 1000 training samples ("attempted-recon" class) and twice when using 2000 training samples ("web-application-attack" and "web-application-activity" classes). When using 2000 training samples, RIPPER performs best in the "web-application-activity" class. The reason for this is that alerts in the sub-sets are randomly chosen, thus a class could have a different number of samples among trials.

2.3 Tests with DS_B

DS_B is used to validate the manual mode and the use of an ad hoc classification. To perform the benchmarks, we use the same n-gram length that achieves the best results in the previous test. Table 6 details our findings for SVM and RIPPER.

Discussion. The test results on DS_B show that Panacea is effective also when using a user-defined classification, regardless of the classification algorithm is chosen. Regarding accuracy rates, RIPPER shows a higher accuracy for most classes, although SVM scores the best classification rate (by a narrow margin).

Only the "buffer overflow" class has a low classification rate. Both algorithms have wrongly classified most of buffer overflow attacks in the "path traversal"

Table 6. Test details (percentage of correctly classified attacks) on DS_B with SVM and RIPPER. RIPPER achieves better accuracy rates for the two most numerous classes, although by a narrow margin. We observe the same trend for the rates reported in Table 5.

Attack Class	SVM	RIPPER
Path Traversal	98.6%	**99.1%**
Cross-site Scripting	97.5%	**98.4%**
SQL Injection	**97.6%**	96.2%
Buffer Overflow	37.5%	37.5%
Percentage of total attacks correctly classified	**98.0%**	97.7%

class. This is because (1) the number of samples is lower than for the other classes, which are at least 10 times more numerous, and 2) a number of the path traversal attacks present some byte encoding that resembles byte values typically used by some buffer overflow attack vectors. In the case of RIPPER, the "path traversal" class has the highest number of samples, hence no rule is induced for it and any non-matching samples is classified in this class.

2.4 Tests with DS_C

An ABS is supposed to detect previously-unknown attacks, for which no signature is available yet. Hence, we need to test how Panacea behaves when the training is accomplished using mostly alerts generated by an SBS but afterwards Panacea processes alerts generated by an ABS. For this final test we simulate the following scenario. A user has manually classified alerts generated by an SBS during the training phase (DS_B) and she uses the resulting model to classify unknown attacks, detected by two different ABSs (POSEIDON and Sphinx). Since we collected few buffer overflow attacks, we use the Sploit framework [37] to mutate some of the original attack payloads and increase the number of samples for this class, introducing attack diversity at the same time. Thus, we obtain additional training samples with a different payload. Table 7 shows the percentage of correctly classified attacks by SVM and RIPPER. For the buffer overflow attacks, we report accuracy values for the original training set (i.e. representing real traffic) and the "enlarged" training set (in brackets).

Discussion. Tests on DS_C show that the SVM performs better than RIPPER when classifying attack instances that have not been observed before. The accuracy rate for the "buffer overflow" class is the lowest, and most of the misclassified attacks have been classified in the "path traversal" class (see the discussion of benchmarks for DS_B). However, with a higher number of training samples (generated by using Sploit), the accuracy rate increases w.r.t. previous tests. This suggest that, with a sufficient number of training samples, Panacea achieves high accuracy rates.

Table 7. Test details (percentage of correctly classified attacks) on DS_C with SVM and RIPPER. SVM perform better than RIPPER in classifying any attack class. For the "buffer overflow" class and the percentage of total attacks correctly classified we report (in brackets) the accuracy rates when Panacea is trained with additional samples generated using the Sploit framework.

Attack Class	SVM	RIPPER
Path Traversal	**98.1%**	94.4%
Cross-site Scripting	**92.6%**	88.9%
SQL Injection	**100.0%**	87.5%
Buffer Overflow	**50.0%** (75.0%)	25.0% (50.0%)
Percentage of total attacks correctly classified	**92.0%** (93.0%)	89.0% (91.0%)

2.5 Summary of Benchmark Results

From the benchmarks results, we can draw some conclusions after having observed the following trends:

- The classification accuracy is always higher than 75%.
- SVM performs better than RIPPER when considering the classification accuracy for all classes, when not all of them have more than 50-60 samples (DS_A, DS_B and DS_C).
- RIPPER performs better than SVM when the class has a good deal of training samples, i.e., at least 60-70 in our experiments (DS_A and DS_B).
- SVM performs better than RIPPER when the class presents high diversity and attacks to classify have not been observed during training (DS_C).

We can conclude that SVM works better when a few alerts are available for training and when attack diversity is high, i.e., the training alert samples differ from the alerts received when in classification phase. On the other hand, RIPPER shows to be more accurate when trained with a high number of alerts.

2.6 System Performance

In Section 1.2 we introduce the requirement of a fast training phase for the classification algorithm. During our benchmarks both SVM and RIPPER proved to satisfy such a requirement. As the BF data size is constant (and it is not related to the n-gram length), the training time depends on the number of alerts processed. Benchmarks have been performed on a machine with an Intel Core 2 CPU at 1.8Ghz and 2Gb of memory. The reported figures refer to benchmarks with DS_A, and are averaged values over five trials. RIPPER is the fastest algorithm and the time required for training grows linearly. When 1000 alerts are used for training, RIPPER completes the training phase in 8.9 seconds and SVM in 11.8 seconds. 3000 alerts are processed in 25.9 seconds by RIPPER and in 39.7 seconds by SVM. While retraining, Panacea can use the old classifier instance

while it builds the new classifier (and the required time is short enough to allow batch processing of a large number of alerts).

A fast classification phase is also desirable, in order to select an appropriate action to handle any alert as soon as it is raised. We report figures for benchmarks with data set DS_C, since for DS_A and DS_B we use the cross-validation testing (where multiple scans of the set are performed). When 1500 alerts are used for training, RIPPER classifies 100 alerts in 0.9 seconds and SVM in 1.3 seconds. Thus, Panacea would be able to process up to 300.000 alerts per hour, a rate that is hardly seen even in large networks.

2.7 Evaluating Confidence

However good Panacea is, the system is not error-free. The consequences of a misclassification can have a direct impact on the overall security. Think of a buffer overflow attack, for which usually countermeasures must take place immediately (because of the possible consequences), that is misclassified as a path traversal attack, for which the activation of countermeasures can be delayed (e.g., after other actions taken by the attacker). This event occurs often in our benchmarks when the system selects the wrong class. Both SVM and RIPPER can generate a classification confidence value for each attack. This value can be used to evaluate the accuracy of the classification. The lower the classification value is (in a range from 0.0 to 1.0), the more likely the classification is wrong (see Table 8 for average confidence values for DS_C).

The confidence value can be taken into consideration to detect possible misclassification. Users can set a minimum confidence value (e.g., 0.5). Any alert with a lower confidence value is forwarded to a human operator for manual classification. With this additional check, we are able to increase the percentage of total attacks correctly classified up to 95% for SVM and 94% for RIPPER (when using the standard training set, without additional training samples generated

Table 8. Effects of confidence evaluation for DS_C, when Panacea is trained with the standard DS_B. When considering the classification confidence to forward alerts for manual classification, the human operator classification increases by 3% and 5% the overall accuracy rate by inspecting 10 and 13 alerts, out of 100, when Panacea uses SVM and RIPPER respectively.

	SVM	**RIPPER**
Average confidence value for correctly classified attacks	0.75	0.62
Average confidence value for misclassified attacks	0.37	0.43
Percentage of total attacks correctly classified without confidence evaluation	92.0%	89.0%
Percentage of total attacks correctly classified with confidence evaluation	95.0%	94.0%
# of alerts forwarded for manual classification	10/100	13/100
# of forwarded attacks that were actually wrongly classified	3/10	5/13
# of forwarded attacks that were actually correctly classified	7/10	8/13

Table 9. Actions that users have to take with or without Panacea w.r.t. alert classification for each data set we use during benchmarks.

	User actions	
	Without Panacea	**With Panacea**
DS_A	Classify any alert	No action to take
DS_B	Classify any alert	Classify alerts used during training
DS_C	Classify any alert	No action to take (alerts have been previously classified)

with Sploit). The additional workload involves also the manual classification of alerts which have been correctly classified by the system but whose confidence value is lower than the set threshold. However, less than 10 alerts (out of 100) have been forwarded for manual classification when this action was not needed. Table 8 reports the details regarding the evaluation of the confidence value.

2.8 Usability in Panacea

Panacea aims not only to provide automatic attack classification for an ABS, but to improve usability as well. In automatic mode, Panacea performs an accurate classification (more than 75% of correctly classified attacks). In semi-automatic and manual modes, users actively take part in the classification process: however, users are requested to provide a limited input (i.e., a class label). Panacea classifies attacks systematically and automates (1) the extraction of relevant information used to distinguish an attack class from another and (2) the update of the classification model. These tasks are usually left to the user's experience and knowledge, thus they can be error-prone and not comprehensive. Table 9 reports actions that users have to take with and without the support of Panacea.

3 Related Work

Although the lack of attack classification is a well-known issue in the field of anomaly-based intrusion detection, little research has been done on this topic.

Robertson et al. [2] suggest to use heuristics to infer the class of (web-based) attacks. This approach has several drawbacks. Users have to generate heuristics (e.g., regular expressions) to identify attack classes. They have to enumerate all of the possible attack variants, and update the heuristics each time a new attack variation is detected. This is a time consuming task. Panacea can operate in an automatic way, by extracting attack information from any SBS, or employ an *ad-hoc* classification, with the user providing only the attack class.

Wang and Stolfo [12] use a "Z-String" to distribute among other ABSs attack payloads to enhance detection. A Z-String contains the information resulting from the n-gram analysis of the attack payload. Once a certain payload has been flagged as malicious, the corresponding Z-String can be distributed to other IDSs to detect the attack also, and stop it at an early stage (think of a worm). If some

traffic matches a certain Z-String, that data is likely to be a real attack. Although a Z-String is not used for attack classification, by attaching a class label it would be possible to classify each attack. However, this approach is not systematic, as each attack that does not exactly match any Z-String would have to be manually classified. A Z-String is based on a frequency-based n-gram analysis, thus an exact match could be difficult to achieve. On the other hand, Panacea applies a systematic classification using the more precise binary-based n-gram analysis. Panacea can also use as a source of information the alerts generated by an SBS, and not only by an ABS.

4 Conclusion

We present Panacea, a system that automatically and systematically classifies attacks detected by a payload-based ABS (and consequently classifies the generated alerts). Panacea extracts information from alerts during a training phase, then predicts the attack class for new alerts. The alerts used to train the classification engine can be generated by an SBS as well as an ABS. In the former case, no manual intervention is requested (the system operates in automatic mode), as Panacea automatically extracts the attack class from the alert. In the latter case, the user is required to provide the attack class for each alert used to train the classification engine.

Panacea improves the usability and makes it possible to integrate anomaly-based with signature-based IDSs. Benchmarks show that the approach is effective in classifying attacks, even those that have not been detected before (and not used for training). Although Panacea works in an automatic way, users can employ ad-hoc classifications, and even manually tune the engine for more precise classifications.

Future work. Panacea can use different algorithms to classify alerts. The benchmarks with SVM and RIPPER, which approach the classification problem in two different ways, show that each algorithm has its strong points, depending on the circumstances. A possible extension is to use a cascade of SVM and RIPPER, in order to increase the overall accuracy. We would then use SVM for early classification (when the number of samples is low, and when RIPPER does not perform well), then, when the number of alerts increases, we can train RIPPER, thanks to the batch training mode, and use it for classification as well (RIPPER performs better than SVM when the number of training samples is high).

References

1. Ghosh, A., Schwartzbard, A.: A study in using neural networks for anomaly and misuse detection. In: SSYM 1999: Proc. 8th conference on USENIX Security Symposium, pp. 141–152. USENIX Association (1999)
2. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.: Using generalization and characterization techniques in the anomaly-based detection of web attacks. In: NDSS 2006: Proc. 13th ISOC Symposium on Network and Distributed Systems Security (2006)

3. Ning, P., Cui, Y., Reeves, D.: Constructing attack scenarios trough correlation of intrusion alerts. In: CCS 2002: Proc. 9th ACM Conference on Computer and Communication Security, pp. 245–254. ACM Press, New York (2002)
4. Cuppens, F., Ortalo, R.: LAMBDA: A Language to Model a Database for Detection of Attacks. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 197–216. Springer, Heidelberg (2000)
5. Debar, H., Wespi, A.: Aggregation and Correlation of Intrusion-Detection Alerts. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 85–103. Springer, Heidelberg (2001)
6. Ning, P., Xu, D.: Learning attack strategies from intrusion alerts. In: CCS 2003: Proc. 10th ACM conference on Computer and Communications Security, pp. 200–209. ACM Press, New York (2003)
7. Valeur, F., Vigna, G., Kruegel, C., Kremmerer, R.: A comprehensive approach to intrusion detection alert correlation. IEEE Trans. Dependable Secur. Comput. 1(3), 146–169 (2004)
8. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: LISA 1999: Proc. 13th USENIX Conference on System Administration, pp. 229–238. USENIX Association (1999)
9. Sourcefire: Snort Network Intrusion Detection System, `http://www.snort.org`
10. Damashek, M.: Gauging similarity with n-grams: Language-independent categorization of text. Science 267(5199), 843–848 (1995)
11. Forrest, S., Hofmeyr, S.: A Sense of Self for Unix Processes. In: S&P 1996: Proc. 17th IEEE Symposium on Security and Privacy, pp. 120–128. IEEE Computer Society Press, Los Alamitos (2002)
12. Wang, K., Stolfo, S.: Anomalous Payload-Based Network Intrusion Detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 203–222. Springer, Heidelberg (2004)
13. Wang, K., Parekh, J., Stolfo, S.: Anagram: a Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
14. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
15. Pietraszek, T.: Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 102–124. Springer, Heidelberg (2004)
16. Bolzoni, D., Crispo, B., Etalle, S.: ATLANTIDES: An Architecture for Alert Verification in Network Intrusion Detection Systems. In: LISA 2007: Proc. 21st Large Installation System Administration Conference, pp. 141–152. USENIX Association (2007)
17. Meyer, D., Leisch, F., Hornik, K.: The support vector machine under test. Neurocomputing 55(1-2), 169–186 (2003)
18. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, `http://www.R-project.org`
19. Lee, W.: A data mining framework for constructing features and models for intrusion detection systems. PhD thesis, Columbia University, New York, NY, USA (1999)
20. Lee, W., Fan, W., Miller, M., Stolfo, S., Zadok, E.: Toward cost-sensitive modeling for intrusion detection and response. Journal of Computer Security 10(1-2), 5–22 (2002)
21. Vapnik, V., Lerner, A.: Pattern recognition using generalized portrait method. Automation and Remote Control 24 (1963)

22. Boser, B., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Proc. 5th Annual ACM Workshop on Computational Learning Theory, pp. 144–152. ACM Press, New York (1992)
23. Cohen, W.: Fast effective rule induction. In: Proc. 12th International Conference on Machine Learning, pp. 115–123. Morgan Kaufmann, San Francisco (1995)
24. The University of Waikato: Weka 3: Data Mining Software in Java, http://www.cs.waikato.ac.nz/ml/weka/
25. Howard, J.: An analysis of security incidents on the Internet 1989-1995. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1998)
26. Hansman, S., Hunt, R.: A taxonomy of network and computer attacks. Computers & Security 24(1), 31–43 (2004)
27. Lippmann, R., Cunningham, R., Fried, D., Garfinkel, S., Gorton, A., Graf, I., Kendall, K., McClung, D., Weber, D., Webster, S., W̃yschogrod, D., Zissman, M.: The 1998 DARPA/AFRL off-line intrusion detection evaluation. In: RAID 1998: Proc. 1st International Workshop on the Recent Advances in Intrusion Detection (1998)
28. Lippmann, R., Haines, J., Fried, D., Korba, J., Das, K.: The 1999 DARPA off-line intrusion detection evaluation. Computer Networks: The International Journal of Computer and Telecommunications Networking 34(4), 579–595 (2000)
29. Snort Team: Snort user manual, http://www.snort.org/docs/snort_htmanuals/htmanual_2832/node220.html
30. Web Application Security Consortium: Web Security Threat Classification, http://www.webappsec.org/projects/threat/
31. Tenable Network Security: Nessus Vulnerabilty Scanner, http://www.nessus.org/
32. CIRT.net: Nikto web scanner, http://www.cirt.net/nikto2
33. Milw0rm, http://milw0rm.com
34. Bolzoni, D., Zambon, E., Etalle, S., Hartel, P.: POSEIDON: a 2-tier Anomaly-based Network Intrusion Detection System. In: IWIA 2006: Proc. 4th IEEE International Workshop on Information Assurance, pp. 144–156. IEEE Computer Society Press, Los Alamitos (2006)
35. Bolzoni, D., Etalle, S.: Boosting Web Intrusion Detection Systems by Inferring Positive Signatures. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part II. LNCS, vol. 5332, pp. 938–955. Springer, Heidelberg (2008)
36. Cova, M., Balzarotti, D., Felmetsger, V., Vigna, G.: Swaddler: An approach for the anomaly-based detection of state violations in web applications. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 63–86. Springer, Heidelberg (2007)
37. Vigna, G., Robertson, W., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. In: CCS 2004: Proc. 11th ACM Conference on Computer and Communications Security, pp. 21–30. ACM Press, New York (2004)

Protecting a Moving Target: Addressing Web Application Concept Drift

Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna

Computer Security Group
UC Santa Barbara
{maggi,wkr,chris,vigna}@cs.ucsb.edu

Abstract. Because of the *ad hoc* nature of web applications, intrusion detection systems that leverage machine learning techniques are particularly well-suited for protecting websites. The reason is that these systems are able to characterize the applications' normal behavior in an automated fashion. However, anomaly-based detectors for web applications suffer from false positives that are generated whenever the applications being protected change. These false positives need to be analyzed by the security officer who then has to interact with the web application developers to confirm that the reported alerts were indeed erroneous detections.

In this paper, we propose a novel technique for the automatic detection of changes in web applications, which allows for the selective retraining of the affected anomaly detection models. We demonstrate that, by correctly identifying legitimate changes in web applications, we can reduce false positives and allow for the automated retraining of the anomaly models.

We have evaluated our approach by analyzing a number of real-world applications. Our analysis shows that web applications indeed change substantially over time, and that our technique is able to effectively detect changes and automatically adapt the anomaly detection models to the new structure of the changed web applications.

Keywords: Anomaly Detection, Web Application Security, Concept Drift, Machine Learning.

1 Introduction

According to a recent study by Symantec [1], web vulnerabilities represent 60% of all reported security flaws. In particular, site-specific vulnerabilities (i.e., those that affect custom web applications) are receiving increased attention from online criminals [2,3]. This is because by exploiting a single vulnerability in a popular site (e.g., a social networking site or a high-traffic portal), an attacker can infect a large number of end hosts by spreading malware via web browser exploits (e.g., drive-by download attacks). Therefore, there is a need for security tools and techniques to protect web applications and deal with their *ad hoc*, dynamic nature.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 21–40, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Anomaly-based intrusion detection techniques have been shown to be effective in protecting web applications against attacks [4,5,6,7,8]. In contrast to misuse detection systems, which contain fingerprints of all *known* attacks patterns, anomaly-based detectors leverage models of the normal behavior of the monitored web applications to detect attacks, under the assumption that attacks cause anomalies, and anomalies are always associated with malicious activity. Besides an initial configuration, these tools typically neither require maintenance nor manual updates to provide protection. For these reasons, they have the advantage of offering a black-box solution to web application security, even against 0-day exploits and site-specific attacks. Some anomaly-based web attack detection techniques are mature enough to be implemented in commercial tools [9,10,11].

A class of anomaly detectors for web applications leverages machine learning techniques to automatically build models of the normal behavior of the monitored web applications. In this context, the term *normal behavior* generally refers to a set of characteristics (e.g., the distribution of the characters of string parameters, the mean and standard deviation of the values of integer parameters) extracted from HTTP messages that are observed during normal operation. Detection is performed under the assumption that attacks cause significant changes (i.e., anomalies) in the application behavior. Thus, any activity that does not fit the expected, learned models is flagged as malicious. Obviously, the detection accuracy strongly depends upon the quality of the models that describe the normal behavior. On one hand, over-specialization can lead to false positives [12,13]; on the other hand, over-generalization often results in false negatives [14,15,16].

One issue that has not been well-studied is the difficulty of adapting to changes in the behavior of the protected applications. By *behavior of a web application*, we refer to the features and the functionalities that the application offers and, as a consequence, the content of the inputs (i.e., the requests) that it process and the outputs (i.e., the responses) that it produces. This is an important problem because today's web applications are user-centric. That is, the demand for new services causes continuous updates to an application's logic and its interfaces.

Our analysis reveals that significant changes in the behavior of web applications are frequent. We refer to this phenomenon as *web application concept drift*. In the context of anomaly-based detection, this means that legitimate behavior might be misclassified as an attack after an update of the application, causing the generation of false positives. Normally, whenever a new version of an application is deployed in a production environment, a coordinated effort involving application maintainers, deployment administrators, and security experts is required. That is, developers have to inform administrators about the changes that are rolled out, and the administrators have to update or re-train the anomaly models accordingly. Otherwise, the amount of false positives will increase significantly. We propose a solution that makes these tedious tasks unnecessary. Our technique examines the responses (HTML pages) sent by a web application. More precisely, we check the forms and links in these pages to determine

when new elements are added or old ones removed. This information is leveraged to identify legitimate changes.

Our technique recognizes when anomalous inputs (i.e., HTTP requests) are due to previous, legitimate updates (changes) in a web application. In such cases, false positives are suppressed by automatically and selectively re-training models. Moreover, when possible, model parameters can be automatically updated without requiring any re-training. Often, a complete re-training would be expensive in terms of time; typically, it requires $O(P)$ where P represents the number of HTTP messages required to train a model. More importantly, such re-training is not always feasible since new, attack-free training data is unlikely to be available immediately after the application has changed. In fact, to collect a sufficient amount of data the new version of the application must be executed and real, legitimate clients have to interact with it in a controlled environment. Clearly, this task requires time and efforts. More importantly, those parts that have changed in the application must be known in advance.

Our approach takes a different perspective. We focus on the fundamental problem of *detecting* those parts of the application that have changed and that will cause false positives if no re-training is performed. Therefore, our technique is agnostic with respect to the specific training procedure, which can be different from the one we propose.

In summary, this paper proposes a set of change detection techniques to address the concept drift problem by treating the protected web applications as oracles. We show that HTTP responses contain important insights that can be effectively leveraged to update previously learned models to take changes into account. The results of applying our technique on real-world data show that learning-based anomaly detectors can automatically *adapt to changes*, and by doing this, are able to reduce their false positive rate without decreasing their detection accuracy.

In this paper, we make the following contributions.

- We detail the problem of concept drift in the context of web applications, and we provide evidence that it occurs in practice, motivating why it is a significant problem for deploying learning-based anomaly detectors in the real world.
- We present novel techniques based on HTTP response models that can be used to distinguish between legitimate changes in web applications and web-based attacks.
- We evaluate a tool incorporating these techniques over an extensive real-world data set, demonstrating its ability to deal with web application concept drift and reliably detect attacks with a low false positive rate.

2 Concept Drift

To introduce the idea of concept drift, we will use a generalized model of learning-based anomaly detectors of web attacks. This model is based on the system presented in [5], but it is general enough to be adapted to virtually any learning-based

anomaly detector for web applications. Also, we show that concept drift is a problem that exists in the real world, and we motivate why it should be addressed. Unless differently stated, we use the shorthand term *anomaly detector* to refer to anomaly-based detectors that leverage unsupervised machine learning techniques.

2.1 Anomaly Detection for Web Applications

An anomaly detector builds models of normal behavior by observing HTTP messages exchanged between servers and clients. The traffic directed to the server running a certain web application (e.g., an e-commerce application or a blog) can be organized into paths, or *resources*, $R = \{r_1, r_2, \ldots, r_j, \ldots\}$. Each resource corresponds to a different software module of the application (e.g., an account manager, a search component). Each resource r_j responds to requests, or *queries*, $Q = \{q_{j,1}, q_{j,2}, \ldots, q_{j,i}, \ldots\}$ that contain sets of name-value parameters transmitted by the client as part of the HTTP request. Each query $q_{j,i}$ is abstracted as a tuple $q_{j,i} = \langle r_j, P_q \rangle$, where $P_q = \{(p_1, v_1), (p_2, v_2), \ldots, (p_k, v_k)\} \subseteq P_j$, and $P_j = P(r_j)$ is the set of *all* the parameters handled by r_j. For instance, the request `'GET /page?id=21&uid=u43&action=del'` contains the resource $r_1 =$ `'/page'` and the parameters $P_q = \{\langle p_1 = \texttt{id}, v_1 = 21 \rangle, \langle p_2 = \texttt{uid}, v_2 = \texttt{'u43'} \rangle, \langle p_3 = \texttt{action}, v_3 = \texttt{'del'} \rangle\}$. Typically, an anomaly detector would use different models to capture legitimate values associated with each parameter.

In addition to requests, the structure of user sessions can be taken into account to model the normal states of a server-side application. In this case, the anomaly detector does not consider individual requests independently, but models their sequence. This model captures the legitimate order of invocation of the resources, according to the application logic. An example is when a user is required to invoke an authentication resource (e.g., `/user/auth`) before requesting a private page (e.g., `/user/profile`). In [5], a session S is defined as a sequence of resources in R. For instance, given $R = \{r_1, r_2, \ldots, r_{10}\}$, a sample session is $S = \langle r_3, r_1, r_2, r_{10}, r_2 \rangle$.

Finally, HTTP responses that are returned by the server can also be modeled. For example, in [5], a model $m^{(\text{doc})}$ is presented that takes into account the structure of documents (e.g., HTML, XML, and JSON) in terms of partial trees that include security-relevant nodes (e.g., `<script />` nodes, nodes containing DOM event handlers, and nodes that contain sensitive data such as credit card numbers). These trees are iteratively merged as new documents are observed, creating a superset of the allowed document structure and the positions within the tree where client-side code or sensitive data may appear.

During the *learning* (or training) phase, given a training set of queries Q and the corresponding responses, the model parameters are estimated and appropriate anomaly thresholds are calculated. More precisely, each parameter of a resource r_i is associated with a set of models; this set of models is called a *profile*: $c_{(\cdot)} = \langle m_1, m_2, \ldots, m_u \rangle$. The specific models in $c_{(\cdot)}$ and the strategy to combine their output determine the classes of attacks that can be detected. The interested reader is referred to [5,8,17] for more details.

During *detection*, for each new request q and corresponding response, the database of profiles is used to calculate an aggregated *anomaly score*, which takes into account the anomaly score of the request or the response according to all the applicable models. In general, an alert is raised if the aggregated anomaly score is above the threshold learned during training.

In this work, the set of models implemented in webanomaly [5] is used to show how anomaly detectors can be improved to cope with the problem of concept drift. However, the techniques we propose in this work can be easily applied to other anomaly-based detectors.

2.2 Web Applications Are Not Static

In machine learning, changes in the modeled behavior are known as *concept drift* [18]. Intuitively, the *concept* is the modeled phenomenon (e.g., the structure of requests to a web server, the recurring patterns in the payload of network packets). Thus, variations in the main features of the phenomena under consideration result in changes, or *drifts*, in the *concept*.

Although the generalization and abstraction capabilities of modern learning-based anomaly detectors are resilient to noise (i.e., small, legitimate variations in the modeled behavior), concept drift is difficult to detect and to cope with [19]. The reason is that the parameters of the models may stabilize to different values. For instance, a string length model could calculate the sample mean and variance of the string lengths that are observed during training. Then, during detection, the Chebyshev inequality is used to detect strings with lengths that significantly deviate from the mean, taking into account the observed variance. Clearly, small differences in the lengths of strings will be considered normal. On the other hand, the mean and variance of the string lengths can completely change because of legitimate and permanent modifications in the web application. In this case, the normal mean and variance will stabilize, or drift, to completely different values. If appropriate re-training or manual updates are not performed, the model will classify benign, new strings as anomalous. This might be a human-intensive activity requiring substantial expertise. Therefore, having an automated, black-box mechanism to adjust the parameters is clearly very desirable.

Changes in web applications can manifest themselves in several ways. In the context of learning-based detection of web attacks, those changes can be categorized into three groups: *request* changes, *session* changes, and *response* changes.

Request changes. Changes in requests occur when an application is upgraded to handle different HTTP requests. These changes can be further divided into two groups: *parameter value* changes and *request structure* changes. The former involve modifications of the actual value of the parameters, while the latter occur when parameters are *added* or *removed*. Parameter *renaming* is the result of removal plus addition.

Example. A new version of a web forum introduces internationalization (I18N) and localization (L10N). Besides handling different languages, I18N and L10N allow several types of strings to be parsed as valid dates and times. For instance,

valid strings for the `datetime` parameter are `'3 May 2009 3:00'`, `'3/12/2009'`, `'3/12/2009 3:00 PM GMT-08'`, `'now'`. In the previous version, valid date-time strings had to conform to the regular expression `'[0-9]{1,2}/[0-9]{2}/[0-9]{4}'`. A model with good generalization properties would learn that the field `datetime` is composed of numbers and slashes, with no spaces. Thus, other strings such as `'now'` or `'3/12/2009 3:00 PM GMT-08'` would be flagged as anomalous. Also, in our example, `tz` and `lang` parameters have been added to take into account time zones and languages. To summarize, the new version introduces two classes of changes. Clearly, the parameter domain of `datetime` is modified. Secondly, new parameters are added.

Changes in HTTP requests directly affect the request models. First, parameter value changes affect any models that rely on the parameters' *values* to extract features. For instance, consider two of the models used in the system described in [5]: $m^{(\mathrm{char})}$ and $m^{(\mathrm{struct})}$. The former models the strings' character distribution by storing the frequency of all the symbols found in the strings during training, while the latter models the strings' structure as a stochastic grammar, using a *Hidden Markov Model* (HMM). In the aforementioned example, the I18N and L10N introduce new, legitimate values in the parameters; thus, the frequency of numbers in $m^{(\mathrm{char})}$ changes and new symbols (e.g., `'-'`, `'[a-zA-Z]'` have to be taken into account. It is straightforward to note that $m^{(\mathrm{struct})}$ is affected in terms of new transitions introduced in the HMM by the new strings. Secondly, request structure changes may affect any type of request model, regardless of the specific characteristics. For instance, if a model for a new parameter is missing, requests that contain that parameter might be flagged as anomalous.

Session changes. Changes in sessions occur whenever resource path sequences are *reordered*, *inserted*, or *removed*. Adding or removing application modules introduces changes in the session models. Also, modifications in the application logic are reflected in the session models as reordering of the resources invoked.

Example. A new version of a web-based community software grants read-only access to *non-authenticated* users, allowing them to display contents previously available to subscribed users only. In the old version, legitimate sequences were ⟨`/site`, `/auth`, `/blog`⟩ or ⟨`/site`, `/auth`, `/files`⟩, where `/site` indicates the server-side resource that handles the public site, `/auth` is the authentication resource, and `/blog` and `/files` were formerly private resources. Initially, the probability of observing `/auth` before `/blog` or `/files` is close to one (since users need to authenticate before accessing private material). This is no longer true in the new version, however, where `/files|/blog|/auth` are all possible after `/site`.

Changes in sessions impact all models that rely on the sequence of resources that are invoked during the normal operation of an application. For instance, consider the model $m^{(\mathrm{sess})}$ described in [5], which builds a probabilistic finite state automaton that captures sequences of *resource paths*. New arcs must be added to take into account the changes mentioned in the above example. These types of models are sensitive to strong changes in the session structure and should be updated accordingly when they occur.

Response changes. Changes in responses occur whenever an application is upgraded to produce different responses. Interface redesigns and feature addition or removal are example causes of changes in the responses. Response changes are common and frequent, since page updates or redesigns often occur in modern websites.

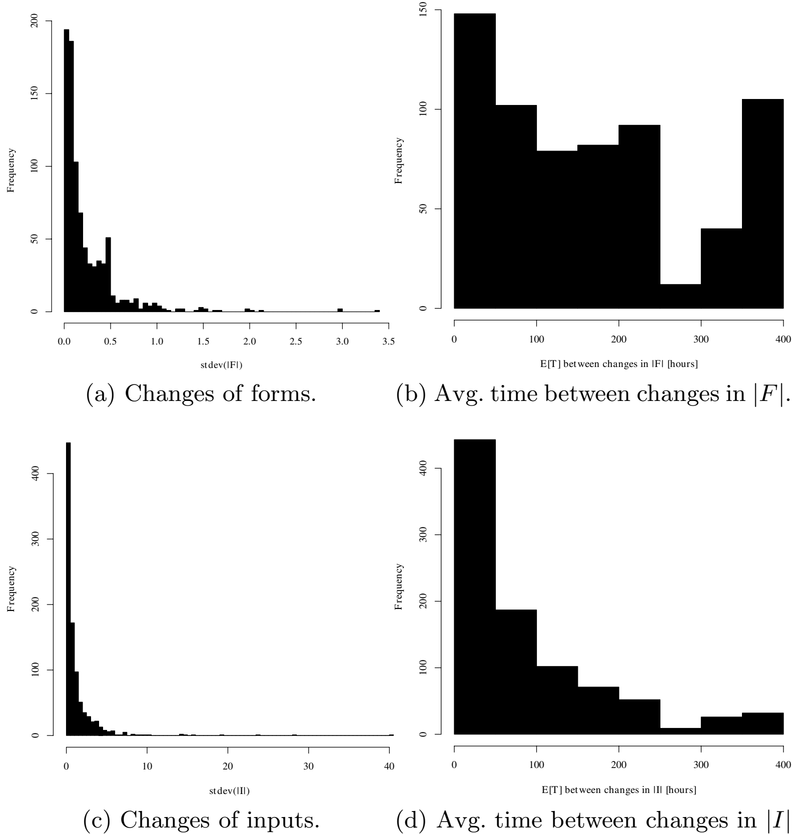
Example. A new version of a video sharing application introduces Web 2.0 features into the user interface, allowing for the modification of user interface elements without refreshing the entire page. In the old version, relatively few nodes of documents generated by the application contained client-side code. In the new version, however, many nodes of the document contain event handlers to trigger asynchronous requests to the application in response to user events. Thus, if a response model is not updated to reflect the new structure of such documents, a large of number of false positives will be generated due to *legitimate* changes in the characteristics of the web application responses.

2.3 Prevalence of Concept Drift

To understand whether concept drift is a relevant issue for real-world websites, we performed three experiments. For the first experiment, we monitored 2,264 public websites, including the Alexa Top 500 and other sites collected by querying Google with popular terms extracted from the Alexa Top 500. The goal was to identify and quantify the changes in the forms and input fields of popular websites at large. This provides an indication of the frequency with which real-world applications are updated or altered.

Once every hour, we visited one representative page for each of the 2,264 websites. In total, we collected 3,303,816 pages, comprising more than 1,390 snapshots for each website, between January 29 and April 13, 2009. One tenth of the representative pages were manually selected to have a significant number of forms, input fields, and hyperlinks with parameters (e.g., `<a href="/login?anon=true&lang=en" />`). By doing this, we gathered a considerable amount of information regarding the HTTP messages generated by some applications. Examples of these pages are registration pages, data submission pages, or contact form pages. For the remaining websites, we simply used their home pages.

For each website w, each page sample crawled at time t is associated with a tuple $|F|_t^{(w)}, |I|_t^{(w)}$, the cardinality of the sets of forms and input fields, respectively. By doing this, we collected samples of the variables $|F|^w = |F|_{t_1}^w, \ldots, |F|_{t_n}^w$, $|I|^w = |I|_{t_1}^w, \ldots, |I|_{t_n}^w$, with $0 < n \lesssim 1,390$. Figure 1 shows the relative frequency of the variables $X_I = \text{stdev}(|I|^{(w_1)}), \ldots, \text{stdev}(|I|^{(w_k)})$ and $X_F = \text{stdev}(|F|^{(w_1)}), \ldots, \text{stdev}(|F|^{(w_k)})$. This demonstrates that a significant amount of websites exhibit variability in the response models, in terms of elements modified in the pages, as well as request models, in terms of new forms and parameters. In addition, we estimated the expected time between changes of forms and inputs fields, $E[T_F]$ and $E[T_I]$, respectively. In terms of forms, 40.72% of the websites drifted during the observation period. More precisely, 922 out of 2,264 websites have a



(a) Changes of forms.

(b) Avg. time between changes in $|F|$.

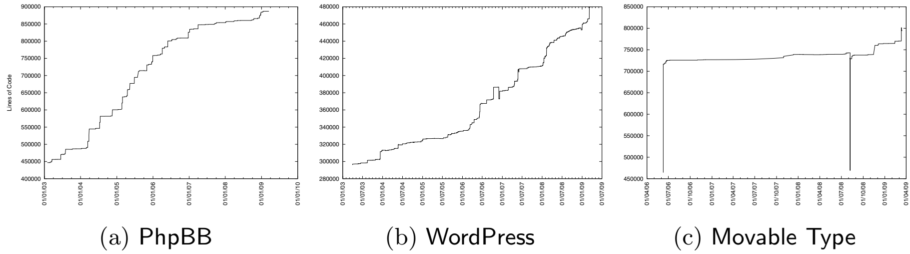
(c) Changes of inputs.

(d) Avg. time between changes in $|I|$

Fig. 1. Relative frequency of the standard deviation of the number of forms (a) and input fields (c). Also, the distribution of the expected time between changes of forms (b) and input fields (d) are plotted. A non-negligible portion of the websites exhibits changes in the responses.

finite $E[T_F]$. Similarly, 29.15% of the websites exhibited drifts in the number of input fields, i.e., $E[T_I] < +\infty$ for 660 websites. Figure 1 shows the relative frequency of (b) $E[T_F]$, and (d) $E[T_I]$. $E[T_F]$. This confirms that a non-negligible portion of the websites exhibit significantly frequent changes in the responses.

For the second experiment, we monitored in depth three large, data-centric web applications over several months: Yahoo! Mail, YouTube, and MySpace. We dumped HTTP responses captured by emulating user interaction using a custom, scriptable web browser implemented with HtmlUnit. Examples of these interactions are as follows: visit the home page, login, browse the inbox, send messages, return to the home page, click links, log out. Manual inspection revealed some major changes in Yahoo! Mail. For instance, the most evident change consisted of a set of new features added to the search engine (e.g., local search, refined address



(a) PhpBB (b) WordPress (c) Movable Type

Fig. 2. Lines of codes in the repositories of PhpBB, WordPress, and Movable Type, over time. Counts include only the code that manipulates HTTP responses, requests and sessions.

field in maps search), which manifested themselves as new parameters found in the web search page (e.g. to take into account the country or the ZIP code). User pages of YouTube were significantly updated with new functionalities between 2008 and 2009. For instance, the new version allows users to rearrange widgets in their personal pages. To account for the position of each element, new parameters are added to the profile pages and submitted asynchronously whenever the user drags widgets within the layout. The analysis on MySpace did not reveal any significant change. The results of these two experiments show that changes in server-side applications are common. More importantly, these modifications often involve the way user data is represented, handled, and manipulated.

For the third experiment, we analyzed changes in the requests and sessions by inspecting the code repositories of three of the largest, most popular open-source web applications: WordPress, Movable Type, and PhpBB. The goal was to understand whether upgrading a web application to a newer release results in significant changes in the features that are used to determine its behavior. In this analysis, we examined changes in the source code that affect the manipulation of HTTP responses, requests, and session data. We used StatSVN, an open-source tool for tracking and visualizing the activity of SVN repositories (e.g., the number of lines changed or the most active developers). We modified StatSVN to incorporate a set of heuristics to compute approximate counts of the lines of code that, directly or indirectly, manipulate HTTP session, request or response data. In the case of PHP, examples representative of such lines include, but are not limited to, `_REQUEST|_SESSION|_POST|_GET|session_|http_|strip_-tags|addslashes`. In order to take into account data manipulation performed through library functions (e.g., WordPress' custom `Http` class), we also generated application-specific code patterns by manually inspecting and filtering the core libraries. Figure 2 shows, over time, the lines of code in the repositories of PhpBB, WordPress, and Movable Type that manipulate HTTP responses, requests and, sessions. These results show the presence of significant modifications in the web application in terms of relevant lines of code added or removed. More

importantly, such modifications affect the way HTTP data is manipulated and, thus, impact request, response or session models.

The aforementioned experiments confirm that the class of changes we described in Section 2.2 is common in real-world web applications. Therefore, we conclude that anomaly detectors for web applications must incorporate procedures to prevent false alerts due to concept drift. In particular, a mechanism is needed to discriminate between legitimate and malicious changes, and respond accordingly.

3 Addressing Concept Drift

In this section, we first present our technique to distinguish between legitimate changes in web application behavior and evidence of malicious behavior. We then discuss how a web application anomaly detection system can effectively handle legitimate concept drift.

3.1 The Web Application as Oracle

The body of HTTP responses contains a set of links L_i and forms F_i that refer to a set of target resources. Each form also includes a set of input fields I_i. In addition, each link $l_{i,j} \in L_i$ and form $f_{i,j} \in F_i$ has an associated set of parameters.

From a resource r_i, the client clicks upon a link $l_{i,j}$ or submits a form $f_{i,j}$. Either of these actions generates a new HTTP request to the web application with a set of parameter key-value pairs, resulting in the return of a new HTTP response to the client, r_{i+1}, the body of which contains a set of links L_{i+1} and forms F_{i+1}. This process continues until the session has ended (i.e., either the user has explicitly logged out, or a timeout has occurred).

Our key observation is that, at each step of a web application session, the set of potential target resources is given exactly by the content of the current resource. That is, given r_i, the associated sets of links L_i and forms F_i directly encode a significant sub-set of the possible r_{i+1}. Furthermore, each link $l_{i,j}$ and form $f_{i,j}$ indicates a precise set of expected parameters and, in some cases, the set of legitimate values for those parameters that can be provided by a client.

Example. Consider a hypothetical banking web application, where the current resource r_i = `/account` presented to a client is an account overview containing a set of links L_i = {`/account/history?aid=328849660322`, `/account/history?aid=446825759916`, `/account/transfer`, `/logout`} and forms (represented as their target action) F_i = {`/feedback`, `/search`}.

From L_i and F_i, we can deduce the set of legal candidate resources for the next request r_{i+1}. Any other resource would, by definition, be a deviation from a legal session flow through the web application as specified by the application itself. For instance, it would not be expected behavior for a client to directly access `/account/transfer/submit` (i.e., a resource intended to submit an account funds transfer) from r_i. Furthermore, for the resource `/account/history`, it is

clear that the web application expects to receive a single parameter aid with an account number as an identifier.

In the case of the form with target /feedback, let the associated input elements be:

```
<select name="subject">
  <option>General</option>
  <option>User interface</option>
  <option>Functionality</option>
</select>
<textarea name="message" />
```

It immediately follows that any invocation of the /feedback resource from r_i should include the parameters subject and message. In addition, the legal set of values for the parameter subject is given by enumerating the enclosed <option /> tags. Similarly, valid values for the new tz and datetime parameters mentioned in the example of Section 2.2 can be inferred. Any deviation from these specifications could be considered evidence of malicious behavior.

We conclude that the responses generated by a web application constitute a specification of the intended behavior of clients and the expected inputs to an application's resources. As a consequence, when a change occurs in the interface presented by a web application, this will be reflected in the content of its responses. Therefore, as detailed in the following section, our anomaly detection system performs response modeling to detect and adapt to changes in monitored web applications.

3.2 Adaptive Response Modeling

In order to detect changes in web application interfaces, the response modeling of webanomaly has been augmented with the ability to build L_i and F_i from the HTML documents returned to a client. The approach is divided into two phases.

Extraction and parsing. The anomaly detector parses each HTML document contained in a response issued by the web application to a client. For each <a /> tag encountered, the contents of the href attribute is extracted and analyzed. The link is decomposed into tokens representing the protocol (e.g., http, https, javascript, mailto), target host, port, path, parameter sequence, and anchor. Paths are subject to additional processing; for instance, relative paths are normalized to obtain a canonical representation. This information is stored as part of an abstract document model for later processing.

A similar process occurs for forms. When a <form /> tag is encountered, the action attribute is extracted and analyzed as in the case of the link href attribute. Furthermore, any <input />, <textarea />, or <select /> and <option /> tags enclosed by a particular <form /> tag are parsed as parameters to the corresponding form invocation. For <input /> tags, the type, name, and value attributes are extracted. For <textarea /> tags, the name attribute is extracted. Finally, for <select /> tags, the name attribute is extracted, as well as the content of any enclosed <option /> tags. The target of the form and its parameters are recorded in the abstract document model as in the case for links.

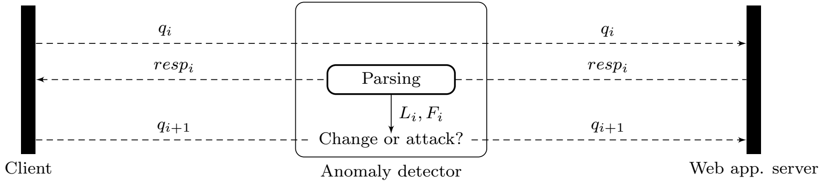


Fig. 3. A representation of the interaction between the client and the web application server, monitored by a learning-based anomaly detector. After request q_i is processed, the corresponding response $resp_i$ is intercepted and link L_i and forms F_i are parsed to update the request models. This knowledge is exploited as a change detection criterion for the subsequent request q_{i+1}.

Analysis and modeling. The set of links and forms contained in a response is processed by the anomaly engine. For each link and form, the corresponding target resource is compared to the existing known set of resources. If the resource has not been observed before, a new model is created for that resource. The session model is also updated to account for a potential transition from the resource associated with the parsed document and the target resource by training on the observed session request sequence.

For each of the parameters parsed from links or forms contained in a response, a comparison with the existing set of known parameters is performed. If a parameter has not already been observed (e.g., the new `tz` parameter), a profile is created and associated with the target resource model.

Any values contained in the response for a given parameter are processed as training samples for the associated models. In cases where the total set of legal parameter values is specified (e.g., `<select />` and `<option />` tags), the parameter profile is updated to reflect this. Otherwise, the profile is trained on subsequent requests to the associated resource.

As a result of this analysis, the anomaly detector is able to adapt to changes in session structure resulting from the introduction of new resources. In addition, the anomaly detector is able to adapt to changes in request structure resulting from the introduction of new parameters and, in a limited sense, to changes in parameter values.

3.3 Advantages and Limitations

Due to the response modeling algorithm described in the previous section, our web application anomaly detector is able to automatically adapt to many common changes observed in web applications as modifications are made to the interface presented to clients. Both changes in session and request structure such as those described in Section 2.2 can be accounted for in an automated fashion. For instance, the I18N and L10N modification of the aforementioned example is correctly handled as it consists in an addition of the `tz` parameter and a modification of the `datetime` parameter. Furthermore, we claim that web application anomaly detectors that do not perform response modeling cannot

reliably distinguish between anomalies caused by legitimate changes in web applications and those caused by malicious behavior. Therefore, as will be shown in Section 4, any such detector that solely monitors requests is more prone to false positives in the real world.

Clearly, the technique relies upon the assumption that the web application has not been compromised. Since the web application, and in particular the documents it generates, is treated as an oracle for whether a change has occurred, if an attacker were to compromise the application in order to introduce a malicious change, the malicious behavior would be learned as normal by our anomaly detector. Of course, in this case, the attacker would already have access to the web application. However, we remark that our anomaly detector observes all requests and responses to and from untrusted clients, therefore, any attack that would compromise response modeling would be detected and blocked. For example, an attacker could attempt to evade the anomaly detector by introducing a malicious change in the HTTP responses and then exploits the change detection technique that would interpret the new malicious request as a legit change. For instance, the attacker could incorporate a link that contain a parameter used to inject the attack vector. To this end, the attacker would have to gain control of the server by leveraging an existing vulnerability[1] of the web application (e.g., a buffer overflow, a SQL injection). However, the HTTP requests used by the attacker to exploit the vulnerability will trigger several models (e.g., the string length model, in the case of a buffer overflow) and, thus, will be flagged as anomalous. In fact, our technique does not alter the ability of the anomaly detector to detect attacks. On the other hand, it avoids many false positives, as demonstrated in Section 4.2.

Besides the aforementioned assumptions, three limitations are important to note. First, the set of target resources may not always be statically derivable from a given resource. For instance, this can occur when client-side scripts are used to dynamically generate page content, including links and forms. Accounting for dynamic behavior would require the inclusion of script interpretation. This, however, has a high overhead, is complex to perform accurately, and introduces the potential for denial of service attacks against the anomaly detection system. For these reasons, we have not included such a component in the current system, although further research is planned to deal with dynamic behavior. Moreover, as Section 4 demonstrates, the proposed technique performs well in practice.

Second, the technique does not fully address changes in the behavior of individual request parameters in its current form. In cases where legitimate parameter values are statically encoded as part of an HTML document, response modeling can directly account for changes in the legal set of parameter values. Unfortunately, in the absence of any other discernible changes in the response, changes in parameter values provided by clients cannot be detected. However, heuristics such as detecting when all clients switch to a new observable behavior in parameter values (i.e., all clients generate anomalies against a set of models in

[1] The threat model assumes that the attacker can interact with the web application only by sending HTTP requests.

a similar way) could serve as an indication that a change in legitimate parameter behavior has occurred.

Third, the technique cannot handle the case where a resource is the result of a parametrized query and the previous response has not been observed by the anomaly detector. In our experience, however, this does not occur frequently in practice, especially for sensitive resources.

4 Evaluation

In this section, we show that our techniques reliably distinguish between legitimate changes and evidence of malicious behavior, and present the resulting improvement in terms of detection accuracy.

The goal of this evaluation is twofold. We first show that concept drift in modeled behavior caused by changes in web applications results in lower detection accuracy. Second, we demonstrate that our technique based on HTTP responses effectively mitigates the effects of concept drift. In both the experiments, the testing data set includes samples of the most common types of attacks against web applications such as cross-site scripting (XSS) (e.g., CVE-2009-0781), SQL injections (e.g., CVE-2009-1224), and command execution exploits (e.g., CVE-2009-0258) that are reflected in request parameter values. In particular, we included a total of 1000 attacks, comprised of 400 XSS attacks, 400 SQL injections, and 200 command injections. The XSS attacks are variations on those listed in [20], the SQL injections were created similarly from [21], and the command execution exploits were variations of common command injections against the Linux and Windows platforms.

In both experiments, webanomaly was evaluated on a data set consisting of HTTP traffic drawn from real-world web applications. This data was obtained from several monitoring points at both commercial and academic sites. For each application, the full contents of each HTTP connection observed over a period of several months were recorded. The resulting flows were filtered using signature-based techniques to remove known attacks, and then partitioned into distinct training and test sets. In total, the data set contains 823 unique web applications, 36,392 unique resource paths, 16,671 unique parameters, and 58,734,624 HTTP requests.

4.1 Effects of Concept Drift

In the first experiment, we demonstrate that concept drift as observed in real-world web applications results in a significant negative impact on false positive rates. First, webanomaly was trained on an unmodified, filtered data set. Then, the detector analyzed a test data set Q to obtain a baseline ROC curve.

After the baseline curve had been obtained, the test data set was processed to introduce new behaviors corresponding to the effects of web application changes, such as upgrades or source code refactoring, obtaining Q_{drift}. In this manner, the set of changes in web application behavior was explicitly known. In particular, as

Table 1. Reduction in the false positive rate due to HTTP response modeling for various types of changes

Change type	Anomalies	False Positives	Reduction
New session flows	6,749	0	100.0%
New parameters	6,750	0	100.0%
Modified parameters	5,785	4,821	16.6%
Total	19,284	4,821	75.0%

detailed in Table 1, 6,749 new session flows were created by introducing requests for new resources and creating request sequences for both new and known resources that had not previously been observed. Also, new parameter sets were created by introducing 6,750 new parameters to existing requests. Finally, the behavior of modeled features of parameter values was changed by introducing 5,785 mutations of observed values in client requests. For example, each sequence of resources ⟨`/login, /index, /article`⟩ might be transformed to ⟨`/login, /article`⟩. Similarly, each request like `/categories` found in the traffic might be replaced with `/foobar`. For new parameters, a set of link or form parameters might be updated by changing a parameter name and updating requests accordingly.

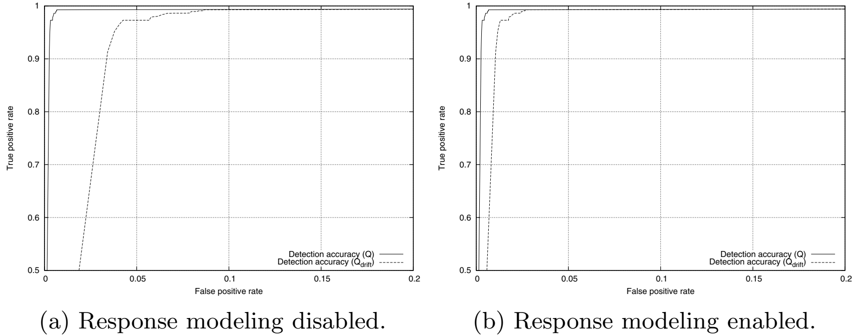
It must be noted that in all cases, responses generated by the web application were modified to reflect changes in client behavior. To this end, references to new resources were inserted in documents generated by the web application, and both links and forms contained in documents were updated to reflect new parameters.

webanomaly – without the HTTP response modeling technique enabled – was then run over Q_{drift} to determine the effects of concept drift upon detector accuracy. The resulting ROC curves are shown in Figure 4a. The consequences of web application change are clearly reflected in the increase in false positive rate for Q_{drift} versus that for Q. Each new session flow and parameter manifests as an alert, since the detector is unable to distinguish between anomalies due to malicious behavior and those due to legitimate change in the web application.

4.2 Change Detection

The second experiment quantifies the improvement in the detection accuracy of webanomaly in the presence of web application change. As before, the detector was trained over an unmodified filtered data set, and the resulting profiles were evaluated over both Q and Q_{drift}. In this experiment, however, the HTTP response modeling technique was enabled.

Figure 4b presents the results of analyzing HTTP responses on detection accuracy. Since many changes in the behavior of the web application and its clients can be discovered using our response modeling technique, the false positive rate for Q_{drift} is greatly reduced over that shown in Figure 4a, and approaches that of Q, where no changes have been introduced. The small observed increase in false positive rate can be attributed to the effects of changes in parameter values. This occurs because a change has been introduced into a parameter value



(a) Response modeling disabled. (b) Response modeling enabled.

Fig. 4. Detection and false positive rates measured on Q and Q_{drift}, with HTTP response modeling enabled in (b)

submitted by a client to the web application, and no indication of this change was detected on the preceding document returned to the client (e.g., because no `<select />` were found).

Table 1 displays the individual contributions to the reduction of the false positive rate due to the response modeling technique. Specifically, the total number of anomalies caused by each type of change, the number of anomalies erroneously reported as alerts, and the corresponding reduction in the false positive rate is shown. The results displayed were generated from a run using the optimal operating point (0.00144, 0.97263) indicated by the knee of the ROC curve in Figure 4b. For changes in session flows and parameters sets, the detector was able to identify an anomaly as being caused by a change in web application behavior in all cases. This resulted in a large net decrease in the false positive rate of the detector with response modeling enabled. The modification of parameters is more problematic, though; as discussed in Section 3.3, it is not always apparent that a change has occurred when that change is limited to the type of behavior a parameter's value exhibits.

From the overall improvement in false positive rates, we conclude that HTTP response modeling is an effective technique for distinguishing between anomalies due to legitimate changes in web applications and those caused by malicious behavior. Furthermore, any anomaly detector that does not do so is prone to generating a large number of false positives when changes do occur in the modeled application. Finally, as it has been shown in Section 2, web applications exhibit significant long-term change in practice, and, therefore, concept drift is a critical aspect of web application anomaly detection that must be addressed.

5 Related Work

Anomaly-based IDSs have evolved considerably after Denning's seminal paper on intrusion detection [22]. Besides network-based detection [23], anomaly-based

techniques have been also exploited to protect the operating system. In [24], the normal behavior of applications is captured by modeling system call sequences [25,26] along with features of their arguments. In [27], a mixture of machine learning techniques is exploited to detect anomalous system calls in the Linux kernel. Ad-hoc distances between system calls are defined to perform clustering in order to identify natural classes of similar calls. The reduced size of the clustered input makes the training of Markov chains efficient. The behavior of each host application is modeled as Markov chains on which probabilistic thresholds are calculated to detect misbehaving sequences.

PAYL [28] is a network-based anomaly detection system. It creates models of each service's normal behavior by recording byte frequencies of network streams. This approach has been further explored in [29], where higher-order n-grams are used instead of frequencies. Instead, [30] exploits self-organizing maps to classify the payload of IP frames in order to separate normal packets from malicious ones.

Anomaly-based detectors of web attacks have been first proposed in [5], where a multi-model approach to characterizing the normal behavior of web application parameters is proposed.

A tool to protect against code-injection attacks has been recently proposed in [17]. The approach exploits a mixture of Markov chains to model legitimate payloads at the HTTP layer. The computational complexity of n-grams with large n is solved using Markov chain factorization, making the system algorithmically efficient.

HTTP responses are exploited in [8]. Besides other features, the DOM is modeled to enhance the detection capabilities of SQL injection and cross-site scripting attacks. The fact that it relies on HTTP responses makes this approach similar to ours. However, we exploit HTTP responses to *detect changes* and update *other* anomaly models accordingly, instead of modeling responses *per se*.

A complementary tool is proposed in [6], where an approach to improve the explanatory power of anomaly-based detectors is proposed along with a clustering and classification methodology to reduce their false positive rate. Another technique to increase detection accuracy is presented in [31], where Bayesian networks are exploited to combine models and define inter-model dependencies. The resulting system shows a significant reduction in false alerts.

Reduction of false positives in anomaly detection systems has also been studied in [13]. Similar behavioral profiles for individual hosts are grouped together using a k-means clustering algorithm. However, the distance metric used was not explicitly defined. Coarse network statistics such as the average number of hosts contacted per hour, the average number of packets exchanged per hour, and the average length of packets exchanged per hour are all examples of metrics used to generate behavior profiles. A voting scheme is used to generate alerts, in which alert-triggering events are evaluated against profiles from other members of that cluster. Events that are deemed anomalous by all members generate alerts.

6 Conclusions

In this work, we have identified the natural dynamicity of web applications as an issue that must be addressed by modern anomaly-based web application anomaly detectors in order to prevent increases in the false positive rate whenever the monitored web application is changed. We refer to this frequent phenomenon the *web application concept drift*.

We propose the use of novel HTTP response modeling techniques to discriminate between legitimate changes and anomalous behaviors in web applications. More precisely, responses are analyzed to find new and previously unmodeled parameters. This information is extracted from anchors and forms elements, and then leveraged to update request and session models. We have evaluated the effectiveness of our approach over an extensive real-world data set of web application traffic. The results show that the resulting system can detect anomalies and avoid false alerts in the presence of concept drift.

As future work, we plan to investigate the potential benefits of modeling the behavior of JavaScript code, which is becoming increasingly prevalent in modern web applications. Also, additional, richer, and media-dependent response models must be studied to account for interactive client-side components, such as Adobe Flash and Microsoft Silverlight applications.

Acknowledgments

The authors wish to thank the anonymous reviewers and our shepherd, Manuel Costa, for their insightful comments. This work has been supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095, and by the European Union though the grant FP7-ICT-216026-WOMBAT.

References

1. Turner, D., Fossi, M., Johnson, E., Mark, T., Blackbird, J., Entwise, S., Low, M.K., McKinney, D., Wueest, C.: Symantec Global Internet Security Threat Report – Trends for July-December 2007. Technical Report XII, Symantec Corporation (April 2008)
2. Shezaf, O., Grossman, J., Auger, R.: Web Hacking Incidents Database (March 2009), http://whid.xiom.org
3. Open Security Foundation: DLDOS: Data Loss Database – Open Source (March 2009), http://datalossdb.org/
4. Cho, S., Cha, S.: SAD: web session anomaly detection based on parameter estimation. In: Computers & Security, vol. 23, pp. 312–319 (2004)
5. Kruegel, C., Robertson, W., Vigna, G.: A Multi-model Approach to the Detection of Web-based Attacks. Journal of Computer Networks 48(5), 717–738 (2005)
6. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.A.: Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2006), San Diego, CA, USA (February 2006)

7. Guangmin, L.: Modeling Unknown Web Attacks in Network Anomaly Detection. In: Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology (ICCIT 2008), Washington, DC, USA, pp. 112–116. IEEE Computer Society, Los Alamitos (2008)
8. Zanero, S., Criscione, C.: Masibty: A Web Application Firewall based on Anomaly Detection. In: DeepSec - In-depth security conference (November 2008)
9. Citrix Systems, Inc.: Citrix Application Firewall (January 2009), http://www.citrix.com/English/PS2/products/product.asp?contentID=25636
10. F5 Networks, Inc.: BIG-IP Application Security Manager (January 2009), http://www.f5.com/products/big-ip/product-modules/application-security-manager.html
11. Breach Security, Inc.: Breach WebDefend (January 2009), http://www.breach.com/products/webdefend.html
12. Axelsson, S.: The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 1999), pp. 1–7. ACM, New York (1999)
13. Frias-Martinez, V., Stolfo, S.J., Keromytis, A.D.: Behavior-Profile Clustering for False Alert Reduction in Anomaly Detection Sensors. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2008), Anaheim, CA, USA (December 2008)
14. Escalante, H.J., Fuentes, O.: Kernel Methods for Anomaly Detection and Noise Elimination. In: Proceedings of the International Conference on Computing (CORE 2006), Mexico City, Mexico, pp. 69–80 (2006)
15. Kim, S.i., Nwanze, N.: Noise-Resistant Payload Anomaly Detection for Network Intrusion Detection Systems. In: Proceedings of the Performance, Computing and Communications Conference (IPCCC 2008), Austin, TX, USA, pp. 517–523. IEEE Computer Society, Los Alamitos (2008)
16. Cretu, G.F., Stavrou, A., Locasto, M.E., Stolfo, S.J., Keromytis, A.D.: Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, CA, USA, pp. 81–95. IEEE Computer Society, Los Alamitos (2008)
17. Song, Y., Stolfo, S., Keromytis, A.: Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In: Proc. of the 16th Annual Network and Distributed System Security Symposium, NDSS (2009)
18. Schlimmer, J., Granger, R.: Beyond incremental processing: Tracking concept drift. In: Proceedings of the Fifth National Conference on Artificial Intelligence, vol. 1, pp. 502–507 (1986)
19. Kolter, J., Maloof, M.: Dynamic weighted majority: An ensemble method for drifting concepts. The Journal of Machine Learning Research 8, 2755–2790 (2007)
20. Hansen, R.: (RSnake): XSS (Cross Site Scripting) Cheat Sheet (June 2009), http://ha.ckers.org/xss.html
21. Mavituna, F.: SQL Injection Cheat Sheet (June 2009), http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
22. Denning, D.E.: An Intrusion-Detection Model. IEEE Transactions on Software Engineering 13(2), 222–232 (1987)
23. Lee, W., Stolfo, S.J.: A Framework for Constructing Features and Models for Intrusion Detection Systems. ACM Transactions on Information and System Security 3(4), 227–261 (2000)
24. Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomaly system call detection. ACM Transactions on Information and System Security 9(1), 61–93 (2006)

25. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P 1996), Oakland, CA, USA, pp. 120–128. IEEE Computer Society, Los Alamitos (1996)
26. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P 2001), Oakland, CA, USA, pp. 156–168. IEEE Computer Society, Los Alamitos (2001)
27. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing 99(1) (5555)
28. Wang, K., Stolfo, S.J.: Anomalous Payload-based Network Intrusion Detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 203–222. Springer, Heidelberg (2004)
29. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
30. Zanero, S.: Analyzing TCP traffic patterns using self organizing maps. In: Roli, F., Vitulano, S. (eds.) ICIAP 2005. LNCS, vol. 3617, pp. 83–90. Springer, Heidelberg (2005)
31. Kruegel, C., Mutz, D., Robertson, W., Valeur, F.: Bayesian Event Classification for Intrusion Detection. In: Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV, USA. IEEE Computer Society, Los Alamitos (2003)

Adaptive Anomaly Detection via Self-calibration and Dynamic Updating

Gabriela F. Cretu-Ciocarlie[1], Angelos Stavrou[2], Michael E. Locasto[2], and Salvatore J. Stolfo[1]

[1] Department of Computer Science, Columbia University
{gcretu,sal}@cs.columbia.edu
[2] Department of Computer Science, George Mason University
{astavrou,mlocasto}@gmu.edu

Abstract. The deployment and use of Anomaly Detection (AD) sensors often requires the intervention of a human expert to manually calibrate and optimize their performance. Depending on the site and the type of traffic it receives, the operators might have to provide recent and sanitized training data sets, the characteristics of expected traffic (*i.e.* outlier ratio), and exceptions or even expected future modifications of system's behavior. In this paper, we study the potential performance issues that stem from fully automating the AD sensors' day-to-day maintenance and calibration. Our goal is to remove the dependence on human operator using an unlabeled, and thus potentially dirty, sample of incoming traffic.

To that end, we propose to enhance the training phase of AD sensors with a self-calibration phase, leading to the automatic determination of the optimal AD parameters. We show how this novel calibration phase can be employed in conjunction with previously proposed methods for training data sanitization resulting in a fully automated AD maintenance cycle. Our approach is completely agnostic to the underlying AD sensor algorithm. Furthermore, the self-calibration can be applied in an online fashion to ensure that the resulting AD models reflect changes in the system's behavior which would otherwise render the sensor's internal state inconsistent. We verify the validity of our approach through a series of experiments where we compare the manually obtained optimal parameters with the ones computed from the self-calibration phase. Modeling traffic from two different sources, the fully automated calibration shows a 7.08% reduction in detection rate and a 0.06% increase in false positives, in the worst case, when compared to the optimal selection of parameters. Finally, our adaptive models outperform the statically generated ones retaining the gains in performance from the sanitization process over time.

Keywords: anomaly detection, self-calibrate, self-update, sanitization.

1 Introduction

In recent years, network anomalies such as flash crowds, denial-of-service attacks, port scans and the spreading of worms and botnets pose a significant threat for large-scale networks. The capability to automatically identify and diagnose anomalous behavior both in the network and on the host is a crucial component of most of the defense and

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 41–60, 2009.
© Springer-Verlag Berlin Heidelberg 2009

failure recovery systems currently deployed in enterprises and organizations. Indeed, Anomaly Detection (AD) sensors are becoming increasingly popular: host-based [24] and network-based [21, 25, 17, 16, 30] intrusion detection systems rely heavily on AD components to maintain their high detection rates and minimize the false positives even when other, non-AD sensors are involved in the detection process.

A major hurdle in the deployment, operation, and maintenance of AD systems is the calibration of these sensors to the protected site characteristics and their ability to "adapt" to changes in the behavior of the protected system. Our aim is to automatically determine the values of the critical system parameters that are needed for both training and long-term operation using only the intrinsic properties of existing behavioral data from the protected host. To that end, we first address the training stage and calibration of the AD sensor. We use an unlabeled, and potentially dirty sample of the training set to construct micro datasets. On one hand, these datasets have to be large enough to generate models that capture a local view of normal behavior. On the other hand, the resulting micro-models have to be small enough to fully contain and minimize the duration of attacks and other abnormalities which will appear in a minority of the micro datasets. To satisfy this trade-off, we generate datasets that contain just enough data so that the arrival rate of new traffic patterns is stable. The micro-models that result from each data set are then engaged in a voting scheme in order to remove the attacks and abnormalities from the data. The voting process is automatically adapted to the characteristics of the traffic in order to provide separation between normal and abnormal data.

The second objective is to maintain the performance level of the AD sensors over a medium or long time horizon, as the behavior of the protected site undergoes changes or evolution. This is not an easy task [21] because of the inherent difficulty in identifying the rate of change over time for a particular site. However, we can "learn" this rate by continuously building new micro-models that reflect the current behavior of the system: every time a new model is added to the voting process, an old model is removed in an attempt to adapt the normality model to the observed changes. Without this adaptation process, legitimate changes in the systems are flagged as anomalous by the AD sensor leading to an inflation of alerts. In contrast, our framework was shown to successfully adapt to modifications in the behavior of the protected system. Finally, our approach is agnostic to the underlying AD sensor, making for a general framework that has the potential to improve the general applicability of AD in the real world.

1.1 Contributions

Our target is to create a fully automated protection mechanism that provides a high detection rate, while maintaining a low false positive rate, and also adapts to changes in the system's behavior. In [5, 4], we have explored the basic problem and proposed the sanitization techniques for multiple sites using empirically determined parameters. We also presented a distributed architecture for coping with long-lasting attacks and a shadow sensor architecture for consuming false positives (FP) with an automated process rather than human attention.

Here, we apply those insights to the problem of providing a run-time framework for achieving the goals stated above. This is a significant advance over our prior work which, while not requiring a manually cleaned data set for training, relied on empirically

determined parameters and human-in-the-loop calibration methods. Along these lines, our current work provides the following contributions:

- Identifying the intrinsic characteristics of the training data, such as the arrival rate of new content and the level of outliers (*i.e.* self-calibration)
- Cleansing a data set of attacks and abnormalities by automatically selecting an adaptive threshold for the voting method presented previously based on the characteristics of the observed traffic resulting in a sanitized training data set (*i.e.* automatic self-sanitization)
- Maintaining the performance we gained by applying the sanitization methods beyond the initial training phase and extending them throughout the lifetime of the sensor by continuously updating the self-calibrated and self-sanitized model (*i.e.* self-update)

2 Ensemble Classifier Using Time-Based Partitions

In [5,4], we focused on methods for sanitizating the training data sets for AD sensors. This resulted in better AD sensor performance (*i.e. higher detection rate while keeping the false positives low*). Here, we attempt to fully automate the construction of those models by calibrating the sanitization parameters using the intrinsic properties of the training data. We briefly describe the sanitization technique and the empirical parameters that it requires in order to operate optimally. Indeed, to cleanse the training data for any AD sensor, we harnessed the idea of an "ensemble classifier", defined by [6] as "a set of classifiers whose individual decisions are combined in some way (typically by weighted or unweighted voting) to classify new examples." One option for generating such an classifier ensemble is to peruse the available training data by splitting them into smaller data sets used to train instances of the AD sensor. The inherent assumption is that *attacks and abnormalities are a minority compared to the entire set of training data*. This is certainly true for training sets that span a long period of time. Therefore, we proposed the use of *time-delimited slices* of the training data. Indeed, consider a large training data set *T* partitioned into a number of smaller disjoint subsets (micro-datasets):

$$T = \{md_1, md_2, \ldots, md_N\}, \tag{1}$$

where md_i is the micro-dataset starting at time $(i-1)*g$ and, g is the granularity for each micro-dataset.

We can now apply a given anomaly detection algorithm. We define the model function *AD* to be:

$$M = AD(T), \tag{2}$$

where *AD* can be any chosen anomaly detection algorithm, *T* is the training data set, and *M* denotes the model produced by *AD* for the given training set. This formulation enables us to maintain the stated principle of being agnostic to the inner workings of the AD sensor - we treat it as a black box whose first task is to output a normality model for a data set provided as input.

We use each of the "epochs" md_i to compute a *micro-model* $M_i = AD(md_i)$ and generate the classifier ensemble. We posit that each distinct attack will be concentrated in (or around) a certain time period, affecting only a small fraction of the micro-models: M_j computed for time period t_j may be poisoned, having modeled the attack vector as normal data, but model M_k computed for time period t_k, $k \neq j$ is likely to be unaffected by the same attack. We use this ensemble classifier for identifying attacks and abnormalities in the data. Our expectation is that the ensemble will be a more efficient tool that the sum of its parts, with the effects of attacks and other abnormalities contained in individual micro-models rather than contaminating the entire data set.

A key parameter of the aforementioned sanitization method is the automatic selection of the optimal time granularity for different training data sets. Intuitively, choring a smaller value of the time granularity g always confines the effect of an individual attack to a smaller neighborhood of micro-models. However, excessively small values can lead to under-trained models that also fail to capture the *normal* aspects of system behavior. One method that ensures that the micro-models are well-trained is based on the rate at which new content appears in the training data [30]. This has the advantage of relying exclusively on intrinsic properties of the training data set. By applying this analysis, we can then identify for each md_i the time granularity that ensures a well-trained micro-model and thus attaining a balance between the two desiderata presented above.

We consider the training data set as a sequence of high-order n-grams (therefore a stream of values from a high-dimensional alphabet). When processing this data, for any time window tw_i, we can estimate the likelihood L_i of the system seeing new n-grams, and therefore new content, in the immediate future based on the characteristics of the traffic seen so far:

$$L_i = \frac{r_i}{N_i}, \tag{3}$$

where r_i is the number of *new unique* n-grams in the time window tw_i and N_i is the *total* number of *unique* n-grams seen between tw_0 and tw_i.

Assuming that the data processed by the system is not random, the value of L_i decreases much faster than the time necessary to exhaust the space of possible n-grams. We are interested in determining the stabilization point for which the number of new grams appears at a low rate, thus looking for the the knee of the curve. In order to detect the stabilization point, we use the linear least squares method over a sliding window of points (in our experiments we use 10 points) to fit a line, $L'_i(t) = a + b * t$. When the regression coefficient b approaches zero (0), we consider that the input has stabilized as long as the standard deviation of the likelihood is not significant. In our experiments, we discovered that we can relax the above assumptions to an absolute value lower than 0.01 for the regression coefficient b while the standard deviation of the likelihood is less than 0.1. The time interval between tw_0 and tw_i is then set as the desired time granularity for computing the micro-models as described above.

Our experimental corpus, used throughout the experiments in this paper, consists of 500 hours of real network traffic from each of two hosts, *www1* and *lists*. *www1* is a gateway to the homepages of students in the Computer Science Department running several dozen different scripts, while *lists* hosts the Computer Science Mailing Lists. The two servers exhibit different content, diversity and volume of data. We partitioned the data into three separate sets: two used for training and one used for testing. The

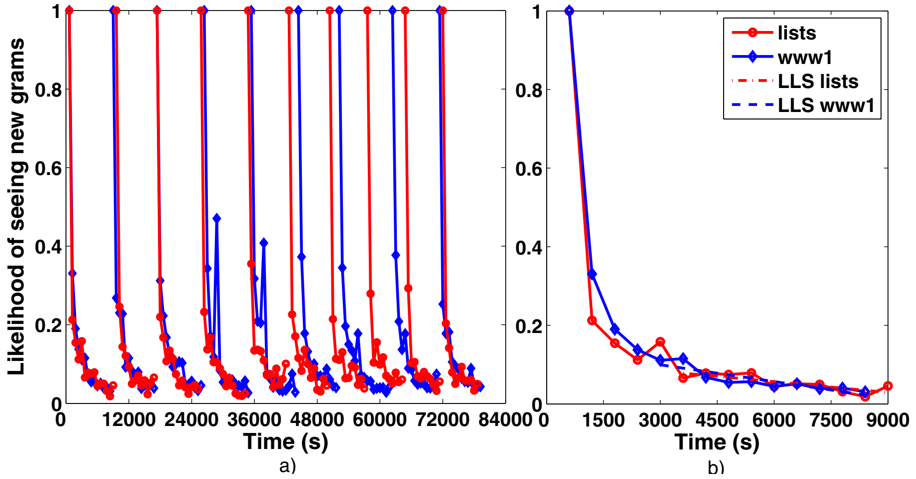


Fig. 1. Time granularity detection($|tw| = 600s$): a) first 10 micro-models (after each model, L is reset); b) zoom on the first model

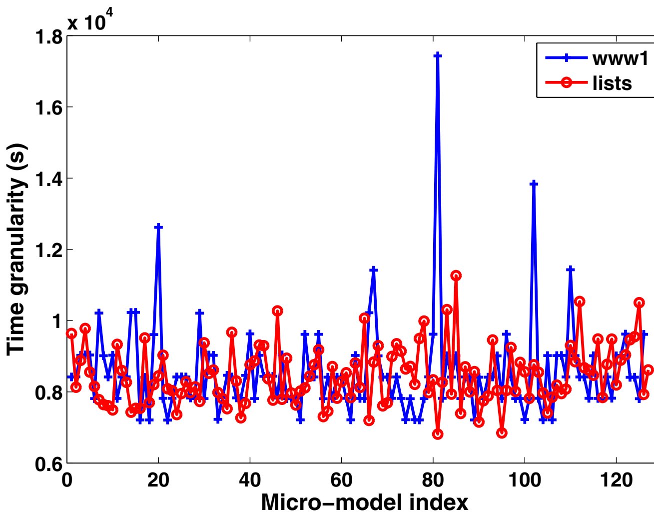


Fig. 2. Automatically determined time granularity

first 300 hours of traffic in each set was used to build micro-models. Figure 1 shows the granularity detection method used to characterize both data sets. Figure 1 (a) presents the time granularity for the first ten micro-models. L is reset immediately after a stabilization point is found, and we begin to generate a new model. At a first glance, both sites display similar behavior, with the level of new content stabilizing within the first few hours of input traffic. However, they do not exhibit the same trend in the likelihood distribution, L_{www1} presenting more fluctuations. Figure 1 (b) presents a zoom on the

first micro-model time granularity detection. The solid lines show the evolution of the L_i likelihood metric over time(we use n-grams of size n=5). The dotted lines show the linear least squares approximation for the stabilization value of tw_i, which is used to compute the time granularity g_i.

Figure 2 illustrates the automatically generated time granularities over the first 300 hours of traffic for both *www1* and *lists*. The average value for *www1* is $g = 8562s$ ($\approx$ 2 hours and 22 minutes), while the standard deviation is $1300s$ ($\approx$ 21 minutes). For *lists* the average time granularity is $g = 8452s$ ($\approx$ 2 hours and 20 minutes), while the standard deviation is $819.8s$ ($\approx$. 13 minutes). In the next section, we will present an extensive comparison between the performance of the sanitized models that use the automated parameters versus the ones built using the empirically determined parameters.

3 Adaptive Training Using Self-sanitization

Once the micro-models are built, they can be used, together with the chosen AD sensor, as a classifier ensemble: a given network packet, which is to be classified as either normal or anomalous, can be tested, using the AD sensor, against each of the micro-models. One possibility would be to apply this testing scheme to the same data set that was used to build the micro-models (we call this process *introspection*). Another option is to apply the micro-model testing to a second set of the initially available traffic, of smaller size. The ultimate goal is to effectively sanitize the training data set and thus obtain the clean training data set needed for anomaly detection.

Once again, we treat the AD sensor at a general level, this time considering a generic $TEST$ function. For a packet P_j part of the tested data set, each individual test against a micro-model results in a label marking the tested packet either as *normal* or *abnormal*:

$$L_{j,i} = TEST(P_j, M_i) \tag{4}$$

where the label, $L_{j,i}$, has a value of 0 if the model M_i deems the packet P_j normal, or 1 if M_i deems it abnormal. However, these labels are not yet generalized; they remain specialized to the micro-model used in each test. In order to generalize the labels, we process each labeled data set through a voting scheme, which assigns a final score to each packet:

$$SCORE(P_j) = \frac{1}{W} \sum_{i=1}^{N} w_i \cdot L_{j,i} \tag{5}$$

where w_i is the weight assigned to model M_i and $W = \sum_{i=1}^{N} w_i$. We have investigated two possible strategies: *simple voting*, where all models are weighted identically, and *weighted voting*, which assigns to each micro-model M_i a weight w_i equal to the number of packets used to train it. In our previous work we observed that the weighted version performs slightly better, so throughout this paper we will use the weighted voting scheme.

The set of micro-models is now ready to be used as an overall packet classifier. Recall our assumption that only a minority of the micro-models will be affected by any given attack or anomaly. Based on the overall score assigned by the set of micro-models, we

split the training data into two disjoint sets: T_{san}, containing the packets deemed as normal, and T_{abn}, containing the abnormalities/attacks:

$$T_{san} = \bigcup \{P_j \mid SCORE(P_j) \leq V\} \tag{6}$$

$$T_{abn} = \bigcup \{P_j \mid SCORE(P_j) > V\}, \tag{7}$$

where V is a *voting threshold* used to differentiate between the two sets. Next we will present our method for automatically computing the value of V that effectively provides this separation, based on the characteristics of the traffic. Once the disjoint data sets are constructed, we can apply the modeling function of the AD sensor and obtain compact representations of both normal and abnormal traffic:

$$M_{san} = AD(T_{san}) \tag{8}$$

$$M_{abn} = AD(T_{abn}) \tag{9}$$

3.1 Voting Threshold Detection

Our goal is to automatically determine the voting threshold, V. In order to establish an effective value for it, we must first analyze the impact of the voting threshold on the number of packets that are deemed normal. The extreme values have an obvious effect: a threshold of $V = 0$ (very restrictive) means that a packet must be approved by all micro-models in order to be deemed normal. In contrast, a threshold of $V = 1$ (very relaxed) means that a packet is deemed as normal as long as it is accepted by at least one micro-model. In general, for a given value V_i we define $P(V_i)$ as the number of packets deemed as normal by the classifier ($SCORE(P_j) < V_i$). The behavior of this function for intermediate values of V_i is highly dependent on the particular characteristics of the available data. For a particular data set, we can plot the function $P(V)$ by sampling the values of V at a given resolution; the result is equivalent to the *cumulative distribution of the classification scores over the entire data set*. This analysis can provide insights into three important aspects of our problem: the intrinsic characteristics of the data (number and relevance of outliers), the ability of the AD sensor to model the differences in the data, and the relevance of the chosen time granularity.

To illustrate this concept, we will use as an example the $www1$ data set and the Anagram [30] sensor. Figure 3 shows the result of this analysis for time granularity ranging from 1 to 100 hours. We notice that, as the time granularity increases, the plot "flattens" towards its upper limit: the classifier loses the ability to discriminate as the micro-models are fewer in number and also more similar between themselves. We also notice that for V very close to 1, all the plots converge to similar values; this is an indicator of the presence of a number of packets that are highly different from the rest of the data in the set.

Intuitively, the optimal voting threshold V is the one that provides the best separation between the normal data class and the abnormal class. The packets that were voted normal for $V = 0$ are not of interest in the separation problem because they are considered normal by the full majority of the micro-models and the choice of V does not influence them. So the separation problem applies to the rest data for which $V > 0$; thus, we normalize $P(V)$ as follows:

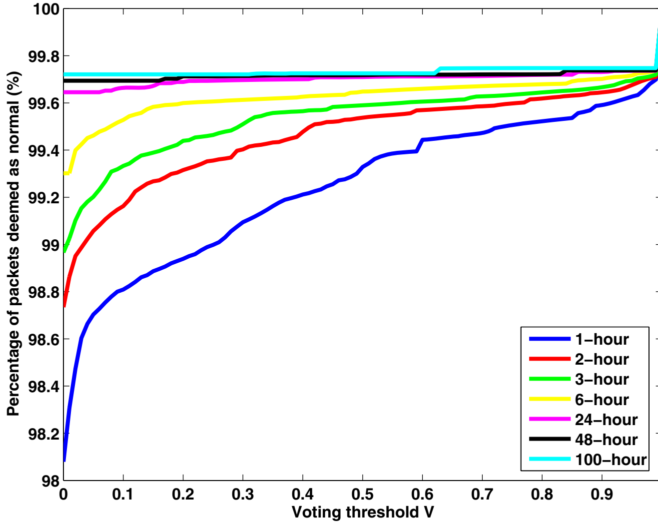


Fig. 3. Impact of the voting threshold over the number of packets deemed as normal for different time granularities

$$p(V_i) = \frac{P(V_i) - P(0)}{P(1) - P(0)} \tag{10}$$

The separation problem can be now considered as the task of finding the smallest threshold (minimize V) that captures as much as possible of the data (maximize $p(V)$). Therefore, if the function $p(V) - V$ exhibits a strong global maximum, these two classes can be separated effectively at the value that provides this maximum.

We have applied this method to both data sets considered in this paper, using Anagram. The profiles of both $p(V)$ (solid lines) and $p(V) - V$ (dotted lines) are shown in Figure 4. In each case, we have marked the value of V that maximizes $p(V) - V$. In both graphs, the maximum of $p(V) - V$ corresponds to a "breaking point" in the profile of $p(V)$ (in general, any changes in the behavior of $p(V)$ are identified by local maxima or minima of $p(V) - V$). The value of the global maximum can be interpreted as a confidence level in the ability of the micro-model classifier to identify outliers, with larger values indicating a high discriminative power between the normal data and the abnormalities/attacks. A low value (and therefore a profile of $p(V)$ following the $x = y$ line) shows that the two classes are not distinct. This can be indicative of a poorly chosen time granularity, an AD sensor that is not sensitive to variations in the data set, or both. We consider this to be a valuable feature for a system that aims towards fully autonomous self-calibration: failure cases should be identified and reported to the user rather than silently accepted.

Once the value of the voting threshold V has been determined, the calibration process is complete. We note that all the calibration parameters have been set autonomously based exclusively on observable characteristics of the training data. The process can therefore be seen as a method for characterizing the combination of AD sensor - training data set, and evaluating its discriminative ability.

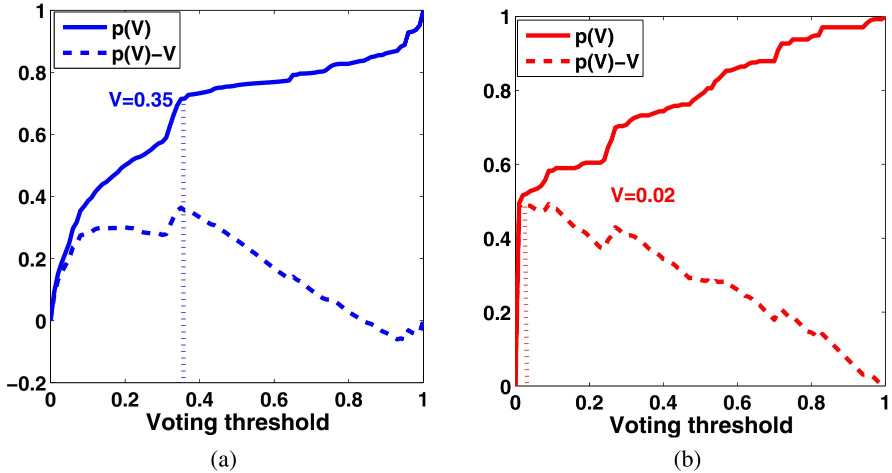


Fig. 4. Determining the best voting threshold for: (a) *www1*; (b) *lists*

3.2 Analysis of Self-sanitization Parameters

To evaluate the quality of the models built using the automatically determined sanitization parameters, we compare their performance against the performance of the sanitized models built using empirically determined parameters. There is a fundamental difference between the two types of models: for the first one the sanitization process is completely hands-free, not requiring any human intervention, while for the latter, exhaustive human intervention is required to evaluate the quality of the models for different parameter values and then to decide on the appropriate parameter values.

There are two parameters of interest in the sanitization process: the set of values for the time granularity and the voting threshold. We will therefore compare the models built using empirically determined parameters against the models built using:

– a fixed time granularity and automatically determined voting threshold;
– automatically determined time granularities and fixed voting threshold;
– both time granularity and voting threshold determined automatically.

Figures 5 and 6 present the false positive and detection rates for models built using different sanitization parameters. The traffic contains instances of phpBB forum attacks (mirela, cbac, nikon, criman) for both hosts that are analyzed.[1] Each line shows the results obtained as the voting threshold was sampled between 0 and 1, with the granularity value either fixed at a given value (usually 1, 3 or 6 hours) or computed automatically using the method described earlier.

[1] Throughout the paper, we refer to detection and false alert rates as rates determined for a specific class of attacks that we observed in these data sets. We note that discovering ground truth for any realistic data set is currently infeasible.

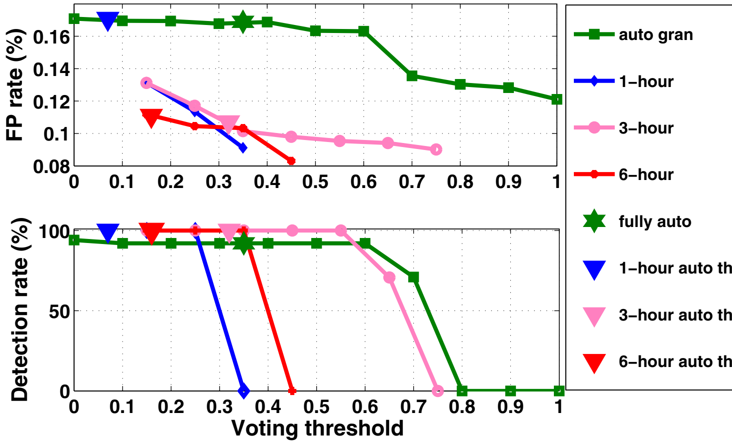


Fig. 5. Model performance comparison for *www1*: automated vs.empirical

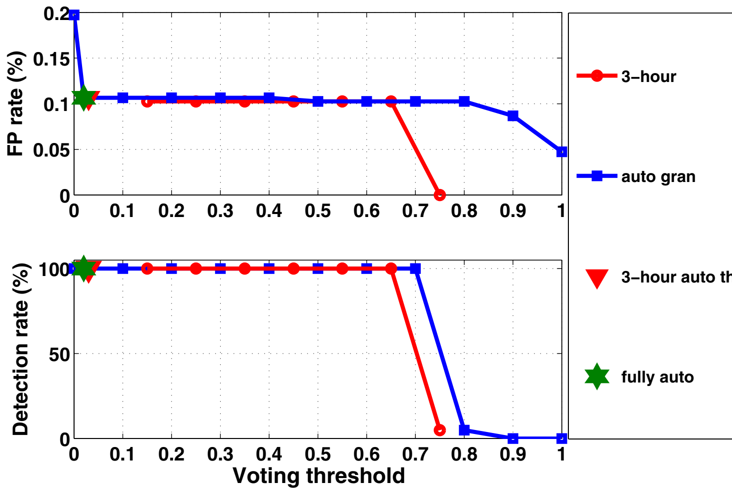


Fig. 6. Model performance comparison for *lists*: automated vs.empirical

We note that the time granularity values empirically found to exhibit high performance were 1-, 3- and 6-hour for *www1*, respectively 3-hour for *lists*. For each of these values, we analyzed the performance of the models built with an automatically determined voting threshold. For each line representing a given granularity value, the triangular markers represent the results obtained with the automatically determined voting threshold. We observe that the voting threshold is placed in the safety zone for which the 100% detection rate is maintained for both *www1* and *lists*, while exhibiting a low false positive rate ($< 0.17\%$).

Table 1. Empirically vs. automatically determined parameters

Parameters	www1		lists	
	FP(%)	TP(%)	FP(%)	TP(%)
N/A(no sanitization)	0.07	0	0.04	0
empirical	0.10	100	0.10	100
fully automated	0.16	92.92	0.10	100

In the case of automated time granularity (the actual values are presented in figure 2), we initially explored the performance of the models determined for different values of the voting threshold, ranging from 0 to 1, with a step of 0.1. In figure 5, for the same fixed threshold, the detection rate is 94.94% or 92.92% compared to the 3-hour granularity (empirical optimal - 100%), while maintaining a low false positive rate ($< 0.17\%$). In figure 6, the results are almost identical to the empirically determined optimal (3-hour granularity).

When we use both the set of time granularities and the voting threshold determined automatically, the system is fully autonomous. In figures 5 and 6, this is indicated by replacing the triangular marker with a star-shaped one. Table 1 also summarizes the values of false positive (FP) and true positive (TP) for the fully automated sanitized model, the empirical optimal sanitized model and the non-sanitized model. With automated parameters, for *lists* we achieve the same values as in the case of empirically determined parameters, while for *www1* the values differ, but we observe that in the absence of the sanitization process the detection rate would be 0. The most important aspect is that the fully-automated sanitization still significantly improves the quality of the AD models while setting its parameters based only on the intrinsic characteristics of the data and without any user intervention.

4 Self-updating Anomaly Detection Models

We presented a method that generates automatically self-sanitized AD models. However, the way users interact with systems can evolve over time [9], as can the systems themselves. As a result, the AD models that once represented the normal behavior of a system can become obsolete over time. Therefor, the models need to adapt to this phenomenon, usually referred to as *concept drift*. As shown in [18], online learning can accommodate changes in the behavior of computer users. Here, we also propose to use an online learning approach to cope with the concept drift, in the absence of ground truth.

Our approach is to continuously create micro-models and sanitized models that incorporate the changes in the data. An aging mechanism can be applied in order to limit the size of the ensemble of classifiers and also to ensure that the most current data is modeled. When a new micro-model, μM_{N+1} is created, the oldest one, μM_1, is no longer used in the voting process (see figure 7). The age of a model is given by the time of its creation.

Every time a new micro-model is generated, a new sanitized model is created as well. In the previous section, we used the micro-models in a voting scheme on a second

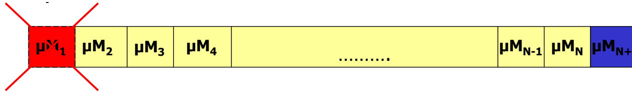


Fig. 7. Incremental learning aging the oldest micro-model

data set, which was processed into a sanitized and an abnormal model. For the online sanitization we will use what we call *introspection*: the micro-models are engaged in a voting scheme against their own micro-datasets[2]. This alternative gives us the ability to apply the self-sanitization processes in an online fashion, without having to also maintain a second dataset strictly for model creation. When a new sanitized model is built, it is immediately used for testing the incoming traffic until a new sanitized model is built.

Concept drift appears at different time scales and our micro-models span a particular period of time. Thus, we are limited in observing drift that appears at scales that are larger than the time window covered by the micro-datasets. Any changes that appear inside this time window are susceptible to being rejected by the voting process rather than being accepted as legitimate evolution of the system. In our online sanitization experiments we use 25 classifiers in the voting process (covering $\approx$ 75 hours of real time traffic) such that we can adapt to drifts that span more than 75 hours of traffic.

We cannot distinguish between a legitimate change and a long-lasting attack that slowly pollutes the majority of the micro-models. A well-crafted attack can potentially introduce malicious changes at the same or even smaller rate of legitimate behavioral drift. As such, it can not be distinguished using strictly introspective methods that examine the characteristics of traffic. However, the attacker has to be aware, guess, or brute-force the drift parameters to be successful with such an attack. In previous work [4], we presented a different type of information that can be used to break this dilemma: alert data from a network of collaborative sites. Another potential solution that we intend to explore as future work, is to employ as feedback information the error responses returned by the system under protection (*e.g. the HTTP reply as an error page*). We plan to explore the conjecture that we can indeed ferret out attacks of certain classes by observing the error responses returned from different sub-systems or software modules.

4.1 Self-update Model Evaluation

To illustrate the self-update modeling, we first apply the online sanitization process for the first 500 hours of traffic using Anagram as the base sensor. Figures 2 and 8 present the fully automated sanitization parameters: the time granularity for each micro-model used in the creation of the new sanitized models, respectively the voting threshold for each newly created sanitized model.

If we didn't employ a model update mechanism, a sanitized model would be built only once. Thus, we call the first sanitized model a *static sanitized model*. Because

[2] We recall that we define a micro-dataset as the training dataset used for building a micro-model.

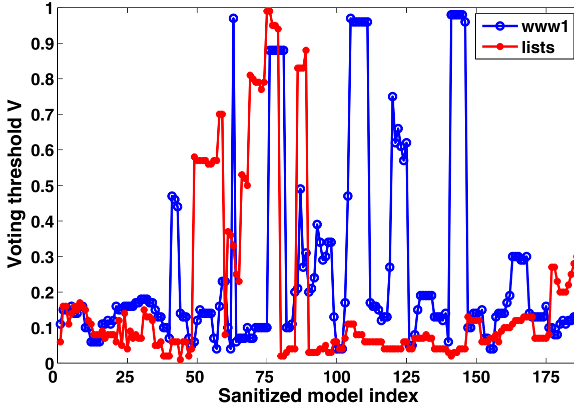


Fig. 8. Automatically determined voting threshold for *www1* and *lists*

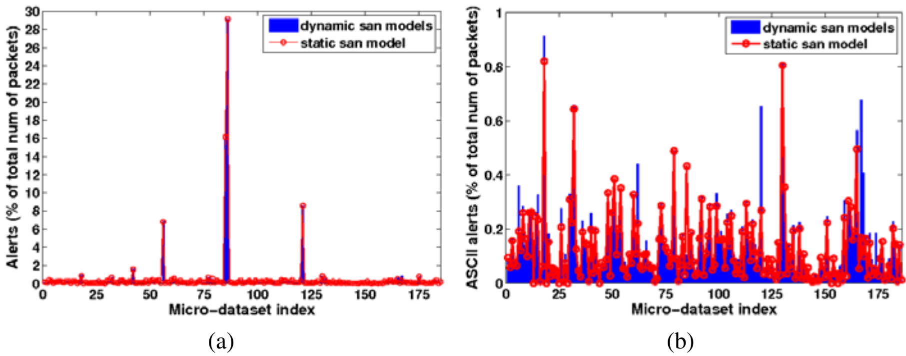


Fig. 9. Alert rate for *www1*: (a) both binary and ascii packets; (b) ascii packets

in the online sanitization process, the models change continuously we consider them *dynamic sanitized models*. To analyze how the online sanitization performs, in figure 9 we compare the static sanitized model alert rate against the dynamic sanitized models alert rate for *www1*.

Figure 9 (a) presents the total number of alerts for each micro-dataset tested with both the static and dynamic models. We first notice that, for a few micro-dates the alert rate reaches levels up to 30% for both model types. After analyzing the alert data, we determined that the high alert rate was generated not by abrupt changes in the system's behavior, but rather by packets containing binary media files with high entropy. This type of data would be considered anomalous by AD sensors such as Anagram. Thus the recommendation is to divert all the media traffic to specialized detectors which can detect malicious content inside binary media files. Figure 9 (b) presents the alert rate after ignoring the binary packets. We can observe that there is no significant difference

Table 2. Static model vs. dynamic models alert rate

Model	www1		lists	
	FP(%)	TP(%)	FP(%)	TP(%)
static model	0.61	94.68	0.13	100
dynamic models	0.62	98.37	0.26	100

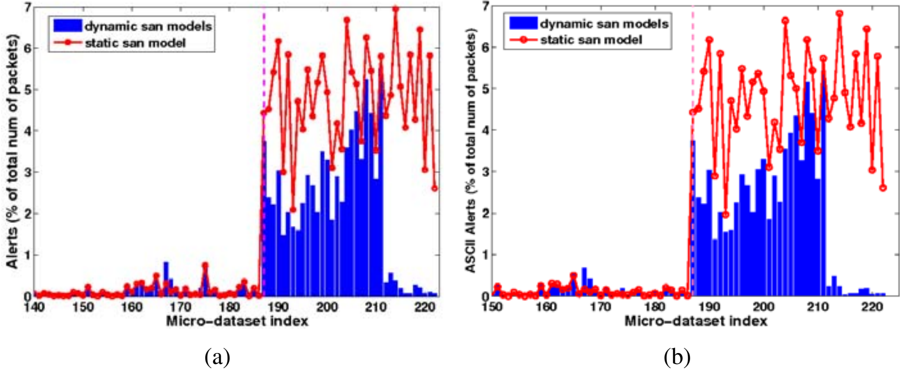


Fig. 10. Concept drift detection for *www1* - alert rate for (a) both binary and ascii packets; (b) ascii packets. Vertical lines mark the boundary between new and old traffic.

between the alert rate exhibited by the static and dynamic sanitized models. Thus we can conclude that there are no fundamental changes over the 500 hour period.

In terms of performance, table 2 presents both the false positive rate (including the binary packets) and the detection rate for *www1* and *lists*. Abrupt changes in the voting threshold (as shown in figure 8) determine the creation of more restrictive models, thus the increase in the detection rate and/or the false positive rate. For *www1* the signal-to-noise ratio (*i.e.* TP/FP) is improved from 155.21 to 158.66, while for *lists* it decreases from 769.23 to 384.61.

We also investigated concept drift appearing at larger scale such as weeks and months, as opposed to days. For this, we tested our method for traffic from the same site, collected at months difference. Figure 10 presents the alert rate for both static and dynamic models, with and without the binary packets. Vertical lines mark the boundary between new and old traffic. We can observe that when changes happen in the system, the alert rate increases for both static and dynamic models. After the dynamic models start updating to the new data, there is a drop in the alert rate, back to levels below 1%. For the static model, the alert rate stays at about 7%, demonstrating the usefulness of a self-updating sanitization process.

Figure 11 presents the raw number of alerts that our system returns on an hourly basis. We note that spikes in the number of alerts can render manual processing difficult, especially when there are changes in the system under protection and the models gradually adapt to the new behavior. However, manual processing of alerts is not the

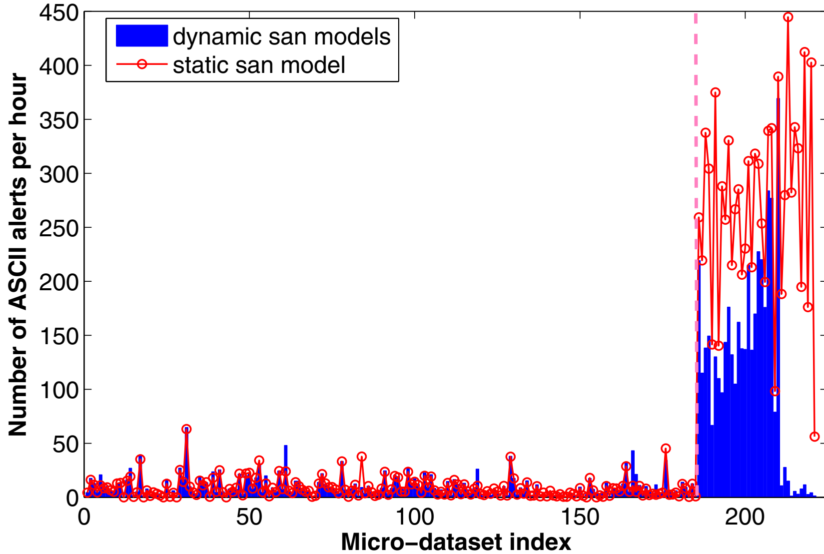


Fig. 11. Number of ASCII alerts per hour for *www1*. The vertical line marks the boundary between new and old traffic.

intended usage model for our framework; our ultimate goal is to build a completely hands-free system that can further identify the true attacks from the false positives. In previous work [4] we have proposed using a shadow sensor architecture such as the ones presented in [22, 1] to automatically consume and validate the false positives. Our study of computational performance presented in [4] shows that, with this architecture, the false positives can be consumed automatically and neither damage the system under protection nor flood an operational center with alarms.

4.2 Computational Performance Evaluation

To investigate the feasibility of our online technique we have to analyze the computational overhead that it implies. Ignoring the initial effort of building the first batch of micro-models and the sanitized model, we are interested in the overhead introduced by the model update process. Table 3 presents a breakdown of the computational stages of this process.

The overhead has a linear dependency on the number and the size of the micro-models. For *www1*, we used 25 micro-models per sanitization process and the size of a micro-model was on average 483 KB (trained on 10.98 MB of HTTP requests). The experiments were conducted on a PC with a 3GHz Intel(R) Xeon(R) CPU with 4 cores and 16G of RAM, running Linux. This level of performance is sufficient for monitoring and updating models on the two hosts that we tested in this paper, as it exceeds the arrival rate of HTTP requests. In the case of hosts displaying higher traffic bandwidth, we can also exploit the intrinsic parallel nature of the computations in order to speed

Table 3. Computational performance for the online automated sanitization for *www1*

Task	Time to process
build and save a new micro-model	7.34 s
test its micro-dataset against the older micro-models	1 m 12 s
test the old micro-datasets against the new micro-model	1 m 58 s
rebuild and save the sanitized model	3 m 03 s

up the online update process: multiple datasets can be tested against multiple models in parallel, as the test for each dataset-model pair is an independent operation. In future work, we will implement a parallel version of this algorithm to test these assumptions.

5 Related Work

We have previously explored the feasibility of sanitizing training datasets using empirically determined parameters [5,4]. This paper presents methods that make the process automatic, by generating the sanitization parameters based only on the intrinsic characteristics of the data and by also coping with concept drift. The sanitization process can be viewed as an ensemble method [6] with the restriction that our work is an unsupervised learning technique. We generate AD models from slices of the training data, thus manipulating the training examples presented to the learning method. Bagging predictors [2] also use a learning algorithm with a training set that consists of a sample of m training examples drawn randomly for the initial data set. ADABoost [11] generates multiple hypothesis and maintains a set of weights over the training example. Each iteration invokes the learning algorithm to minimize the weighted error and returns a hypothesis, which is used in a final weighted vote.

MetaCost [7] is an algorithm that implements cost-sensitive classification. Instead of modifying an error minimization classification procedure, it views the classifier as a black box, the same as we do, and wraps the procedure around it in order to reduce the loss. MetaCost estimates the class probabilities and relabels the training examples such that the expected cost of predicting new labels is minimized. Finally it builds a new model based on the relabeled data. JAM [27] focuses on developing and evaluating a range of learning strategies for fraud detection. That work presents methods for "meta-learning" by computing sets of "base classifiers" over various partitions or sampling of the training data. The combining algorithms proposed are called "class-combiner" or "stacking" and they are built based on work presented in [3] and [31]. For more details on meta-learning techniques we can also refer the reader to a more comprehensive survey [23].

The perceived utility of anomaly detection is based on the assumption that malicious inputs rarely occur during the normal operation of the system. Because a system can evolve over time, it is also likely that new *non-malicious* inputs will be seen [10]. Perhaps more troubling, Fogla and Lee [8] have shown how to evade anomaly classifiers by constructing polymorphic exploits that blend with normal traffic (a sophisticated form of mimicry attack [28]), and Song *et al.* [26] have improved on this technique and

shown that content–based approaches may not work against all polymorphic threats, since many approaches often fix on specific byte patterns [19].

The problem of determining anomaly detection parameters have been studied before. Anagram [30] determines the model stability automatically based on the rate at which new content appears in the training data. pH [24] proposes heuristics for determining an effective training time, minimizing the human intervention as well. Payl [29] has a calibration phase for which a sample of test data is measured against the centroids and an initial threshold setting is chosen. The thresholds are updated throughout a subsequent round of testing. In [17], the authors propose a web-based anomaly detection mechanism, which uses a number of different models to characterize the parameters used in the invocation of the server-side programs. For these models, dynamic thresholds are generated in the training phase, by evaluating the maximum score values given on a validation dataset. PCA-based techniques for detecting anomalous traffic in IP networks became popular in the past years. [21] talks about the difficulty of tuning the parameters for these techniques and discusses pollution of the normal subspace.

The concept of updating an AD sensor in order to mirror valid changes in the protected system's behavior is discussed in [18]. Most publications which propose updating the model after significant changes to the environment, data stream, or application use supervised learning techniques, such as [12]. Methods of this type maintain an adaptive time window on the training data [14], select representative training examples [13], or weigh the training examples [15]. The key idea is to automatically adjust the window size, the example selection, and the example weighting, respectively, so that the estimated generalization error is minimized. Consequently, these methods assume the existence of labeled data which is not the case for the applications that we interested in analyzing. It seems that anomaly detectors would benefit from an additional source of information that can confirm or reject the initial classification, and Pietraszek [20] suggests using human–supervised machine learning for such tuning.

6 Conclusions and Future Work

Anomaly detection sensors have become an integral part of the network and host-based defenses both for large-scale network and individual users. Currently, AD sensors require human operators to perform initial calibration of the training parameters to achieve optimal detection performance and minimize the false positives. In addition, as the protected system evolves over time, the sensor's internal state becomes more and more inconsistent with the protected site. This discrepancies between the initial normality model and the current system behavior eventually renders the AD sensor unusable.

To amend this, we propose a fully automated framework that allows the AD sensor to adapt to the characteristics of the protected host during the training phase. Furthermore, we provide an online method to maintain the state of the sensor, bounding the deviations due to content or behavioral modifications that are consistent over a period of time. Without this adaptation process and the generation of new normality models which we call "dynamic", legitimate changes in the systems are flagged as anomalous by the AD sensor leading to an inflation of alerts. Our experimental results show that, compared to the manually obtained optimal parameters, the fully automated calibration has either

identical, or slightly reduced (by 7.08%) detection rate and a 0.06% increase in false positives. Furthermore, over a very large time window, our dynamic model generation maintains a low alert rate (1%) as opposed to a 7% for a system without updates.

We believe that our system can help alleviate some of the challenges faced as anomaly detection is increasingly relied upon as a first-class defense mechanism. AD sensors can help counter the threat of zero-day and polymorphic attacks; however, the reliance on user input is a potential roadblock to their application outside of the lab and into commercial off-the-shelf software. In this paper we have taken a number of steps towards AD sensors that enable true hands-free deployment and operation.

In the future, we intend to establish this feature of our framework by using more sensors, that either model data in a different way (*e.g.* Payl [29], libanomaly [17], Spectrogram [25]) or target different applications (*e.g.* pH [24]). Despite the best efforts of the research community, no AD sensor has been proposed to date that can detect all attack types while maintaining a low alert rate. A possible option, which we intend to further explore in the future, is to combine the strengths of multiple sensors under a general and unified framework, following the directions traced out in this study.

Finally, the methods presented harness the information contained in the traffic (or behavior in general) of the protected host. Large-scale implementations of AD systems can further benefit by exchanging data, such as micro-models or sanitized and abnormal models, across different sites. Therefore, the temporal dimension of our online sanitization process can be complemented by a spatial one. We are currently in the process of establishing an information exchange framework that can facilitate these experiments; we plan to report these result in a future study.

Acknowledgments

This material is based on research sponsored by the Air Force Office of Scientific Research AFOSR MURI under contract number GMU 107151AA, by the National Science Foundation under NSF grants CNS-06-27473 and by Google INC. We authorize the U.S. Government to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Anagnostakis, K.G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., Keromytis, A.D.: Detecting Targeted Attacks Using Shadow Honeypots. In: Proceedings of the 14th USENIX Security Symposium (2005)
2. Breiman, L.: Bagging Predictors. Machine Learning 24(2), 123–140 (1996)
3. Chan, P.K., Stolfo, S.J.: Experiments in Multistrategy Learning by Meta-Learning. In: Proceedings of the second international conference on information and knowledge management, Washington, DC, pp. 314–323 (1993)
4. Cretu, G.F., Stavrou, A., Locasto, M.E., Stolfo, S.J., Keromytis, A.D.: Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In: The Proceedings of the IEEE Symposium on Security and Privacy (2008)

5. Cretu, G.F., Stavrou, A., Stolfo, S.J., Keromytis, A.D.: Data Sanitization: Improving the Forensic Utility of Anomaly Detection Systems. In: Workshop on Hot Topics in System Dependability, HotDep (2007)
6. Dietterich, T.G.: Ensemble Methods in Machine Learning. In: Kittler, J., Roli, F. (eds.) MCS 2000. LNCS, vol. 1857, pp. 1–15. Springer, Heidelberg (2000)
7. Domingos, P.: Metacost: A general method for making classifiers cost-sensitive. In: Knowledge Discovery and Data Mining, pp. 155–164 (1999)
8. Fogla, P., Lee, W.: Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), pp. 59–68 (2006)
9. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: IEEE Symposium on Security and Privacy (1996)
10. Forrest, S., Somayaji, A., Ackley, D.: Building Diverse Computer Systems. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems, pp. 67–72 (1997)
11. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: European Conference on Computational Learning Theory, pp. 23–37 (1995)
12. Gama, J., Medas, P., Castillo, G., Rodrigues, P.P.: Learning with drift detection. In: XVII Brazilian Symposium on Artificial Intelligence (2004)
13. Klinkenberg, R.: Meta-learning, model selection, and example selection in machine learning domains with concept drift. In: Learning – Knowledge Discovery – Adaptivity (2005)
14. Klinkenberg, R., Joachims, T.: Detecting concept drift with support vector machines. In: The Proceedings of the 17th Int. Conf. on Machine Learning (2000)
15. Klinkenberg, R., Ruping, S.: Concept drift and the importance of examples. In: Franke, J., Nakhaeizadeh, G., Renz, I. (eds.) Text Mining Theoretical Aspects and Applications (2003)
16. Kruegel, C., Toth, T., Kirda, E.: Service Specific Anomaly Detection for Network Intrusion Detection. In: Symposium on Applied Computing (SAC), Madrid, Spain (2002)
17. Kruegel, C., Vigna, G.: Anomaly Detection of Web-based Attacks. In: ACM Conference on Computer and Communication Security, Washington, DC (2003)
18. Lane, T., Broadley, C.E.: Approaches to online learning and concept drift for user identification in computer security. In: 4th International Conference on Knowledge Discovery and Data Mining (1998)
19. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically Generating Signatures for Polymorphic Worms. In: IEEE Security and Privacy, Oakland, CA (2005)
20. Pietraszek, T.: Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 102–124. Springer, Heidelberg (2004)
21. Ringberg, H., Soule, A., Rexford, J., Diot, C.: Sensitivity of pca for traffic anomaly detection. In: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp. 109–120. ACM, New York (2007), http://doi.acm.org/10.1145/1254882.1254895
22. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a Reactive Immune System for Software Services. In: Proceedings of the USENIX Technical Conference (2005)
23. Smith-Miles, K.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. 41(1) (2008), http://dblp.uni-trier.de/db/journals/csur/csur41.html#Smith-Miles08
24. Somayaji, A., Forrest, S.: Automated Response Using System-Call Delays. In: Proceedings of the 9th USENIX Security Symposium (2000)
25. Song, Y., Keromytis, A.D., Stolfo, S.J.: Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium, NDSS (2009)

26. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the Infeasibility of Modeling Polymorphic Shellcode. In: ACM Computer and Communications Security Conference, CCS (2007)
27. Stolfo, S., Fan, W., Lee, W., Prodromidis, A., Chan, P.: Cost-based Modeling for Fraud and Intrusion Detection: Results from the JAM Project. In: Proceedings of the DARPA Information Survivability Conference and Exposition, DISCEX (2000)
28. Wagner, D., Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems. In: ACM CCS (2002)
29. Wang, K., Cretu, G., Stolfo, S.J.: Anomalous Payload-based Worm Detection and Signature Generation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 227–246. Springer, Heidelberg (2006)
30. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
31. Wolpert, D.: Stacked Generalization. Neural Networks 5, 241–259 (1992)

Runtime Monitoring and Dynamic Reconfiguration for Intrusion Detection Systems

Martin Rehák[1], Eugen Staab[2], Volker Fusenig[2], Michal Pěchouček[1], Martin Grill[3,1], Jan Stiborek[1], Karel Bartoš[3,1], and Thomas Engel[2]

[1] Department of Cybernetics, Czech Technical University in Prague
{rehak,pechoucek,grill,stiborek,bartos}@agents.felk.cvut.cz
[2] Faculty of Science, Technology and Communication, University of Luxembourg
{eugen.staab,volker.fusenig}@uni.lu
[3] CESNET, z. s. p. o., Prague, Czech Republic

Abstract. Our work proposes a generic architecture for runtime monitoring and optimization of IDS based on the challenge insertion. The challenges, known instances of malicious or legitimate behavior, are inserted into the network traffic represented by NetFlow records, processed with the current traffic and the system's response to the challenges is used to determine its effectiveness and to fine-tune its parameters. The insertion of challenges is based on the threat models expressed as attack trees with attached risk/loss values. The use of threat model allows the system to measure the expected undetected loss and to improve its performance with respect to the relevant threats, as we have verified in the experiments performed on live network traffic.

1 Introduction

One of the principal problems of the intrusion detection systems based on the anomaly detection [1] principles is their error rate, both in terms of false negatives (undetected attacks) and false positives, i.e. legitimate traffic labeled as malicious. This problem is amplified by the fact that the sensitivity (and consequently the error rate) varies dynamically as a function of the background traffic. For example, an attack that would be easily discovered in the lower nighttime traffic will pass undetected during the day, on the system with identical settings. In this work, we address the problem of correct IDS monitoring and dynamic reconfiguration, in order to provide the operators with:

- an estimate of system sensitivity/error rate, given the current network traffic and a threat model, and
- autonomous system reconfiguration, based on the system monitoring and the threat model.

In order to perform these tasks, we use the concept of challenges [2] (or fault injection) from the field of autonomic computing, which allows us to measure the response of the system with respect to a small subset of *challenges*, known instances of malicious or legitimate behavior, inserted into the traffic observed on the network. The response of the system and its individual components to the inserted challenges is used to determine

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 61–80, 2009.
© Springer-Verlag Berlin Heidelberg 2009

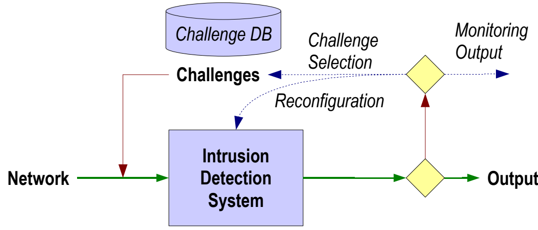


Fig. 1. Adaptation process overview

its current error rate in terms of estimated ratio of false positives/false negatives (see Fig. 1). It is also used to adapt the system behavior and to select and/or create optimal system settings.

This generic concept is verified by its integration with the CAMNEP intrusion detection system [3][4], which is based on a multi-stage combination of several network behavior analysis algorithms processing the NetFlow [5] data. In Section 2, we briefly discuss the relevant properties of the CAMNEP system, which was augmented with the processes described in this paper. Then, we present the self-adaptive architecture integrated with the underlying system and discuss the crucial elements of the architecture (Section 3), such as dynamic classifier selection and optimization of number of challenges and their composition. These sections describe the core contribution of this work.

2 CAMNEP System

The self-optimization techniques presented in this paper were integrated with the CAMNEP network intrusion detection system [3], based on the Network Behavior Analysis (NBA) approach [6]. This system processes NetFlow/IPFIX data provided by routers or other network equipment and uses this information to identify malicious traffic by means of collaborative, multi-algorithm anomaly detection. The system uses the multi-algorithm and multi-stage approach to optimize the error rate, while not compromising the performance of the system. The self-monitoring and self-adaptation techniques are very relevant in this context, as they allow to improve the error rate with only a minimal and controllable impact on its efficiency.

The NetFlow network traffic data is structured in records, and each record describes one *flow*. A flow can be described as an unidirectional component of TCP connection (or its UDP/ICMP equivalent) and contains all packets with the same source IP, destination IP, source and destination port and protocol (TCP/UDP/ICMP). A flow record contains this basic information, as well as other information, such as the number of packets/bytes transferred, duration and TCP flags encountered in the packets of the flow. The flow records are aggregated over a predefined observation period (typically 1-5 minutes). When the observation period elapses, the data is read out for analysis, and a new observation period begins.

The system contains two principal classes of classifying agents, which are able to evaluate the received traffic:

Detection agents (agents A and B in Fig. 2) analyze raw network flows by their anomaly detection algorithms, exchange the anomalies between them and use the aggregated anomalies to build and update the long-term anomaly associated with the abstract traffic classes built by each agent. These traffic classes describe various behaviors, as they can be distinguished based on the features used by the anomaly detection methods integrated into the system. Each detection agent uses one of the five anomaly detection methods listed herein. Each of the methods works with a different traffic model based on a specific combination of aggregate traffic features, such as: (*i*) entropies of flow characteristics for individual source IP addresses [7], (*ii*) deviation of flow entropies from the PCA-based prediction model of individual sources [8], (*iii*) deviation of traffic volumes from the PCA-based prediction for individual major sources [9], (*iv*) rapid surges in the number of flows with given characteristics from the individual sources [10] and (*v*) ratios between the number of destination addresses and port numbers for individual sources [11].

All detection agents map the same flows, together with the shared evaluation of these events, the aggregated immediate anomaly of these events determined by their anomaly detection algorithms, into the traffic clusters built using different features/metrics, thus building the aggregate anomaly hypothesis based on different premises. The *aggregated anomalies* associated with the individual traffic classes are built and maintained using the classic trust modeling techniques (not to be confused with the way trust is used in this work). The detection agents evaluate the anomaly of each network flow on the whole [0,1] interval, and the output of the detection agents is integrated by the aggregation agents.

Aggregation agents α_1 from the set $A = \{\alpha_1, \dots, \alpha_g\}$ represent the various aggregation operators used to build the joint conclusion regarding the normality/anomaly of the flows from the individual opinions provided by the detection agents. Each agent uses a distinct averaging operator (based on order-weighted averaging [12] or simple weighted averaging) to perform the $R^{g_{det}} \rightarrow R$ transformation from the g_{det}-dimensional space to a single real value, thus defining one composite system output that integrates the results of several detection agents. The aggregation agents also dynamically determine the threshold values used to transform the continuous aggregated anomaly value in the $[0, 1]$ interval into the crisp normal/anomalous assessment for each flow. The value of the threshold is either relative (i.e. leftmost part of the distribution) or absolute, based on the evaluation of the agent's response to challenges.

The detection and aggregation agents annotate the individual flows φ with a continuous *anomaly/normality* value in the $[0, 1]$ interval, with the value 1 corresponding to perfectly normal events and the value 0 to completely anomalous ones. This continuous anomaly value describes an agent's opinion regarding the anomaly of the event, and the agents apply adaptive or predefined thresholds to split the $[0, 1]$ interval into the normal and anomalous classes. The threshold applied (and dynamically maintained) by the aggregation agents divides the flows into two classes: *normal* and *anomalous*. The anomalous flows are those whose anomaly falls below the threshold, while the normal

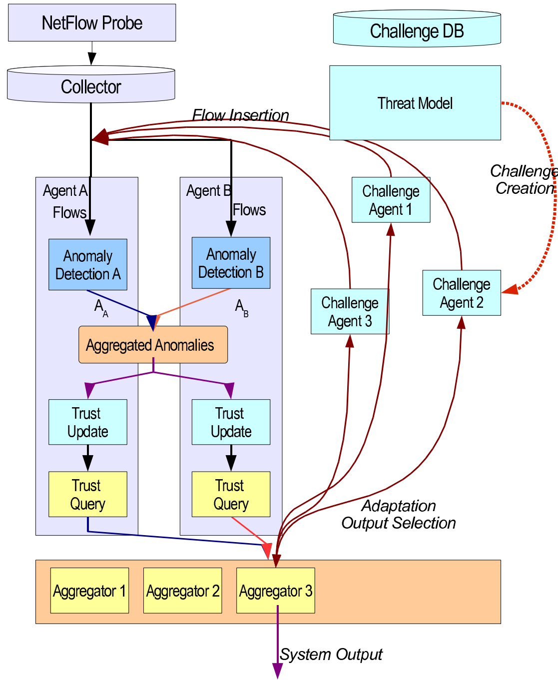


Fig. 2. Adaptation process in the CAMNEP system

flows are those, whose anomaly is above the threshold. This distinction allows us to introduce the components of the error rate. *False Positives* (FP) are the legitimate flows classified as anomalous, while the *False Negatives* (FN) are the malicious flows classified as normal. Most standalone NBA methods suffer from a very high rate of false positives, which makes them unpractical for deployment. The static multi-stage process of the original CAMNEP system already removes a large part of false positives, while not increasing the rate of false negatives, and the goal of the self-optimization techniques is to further improve the effectiveness of the system.

3 IDS Monitoring Architecture

The monitoring and adaptation components of the CAMNEP system implement the high-level functional schema introduced in Fig. 1. The reconfiguration action (as shown in Fig. 1) is the identification of the optimal anomaly aggregation function that achieves the best separation between the legitimate and malicious challenges. Assuming that these challenges are representative of the traffic in the network and the expected attacks, such aggregation should also optimize the performance against the actual threats in the current network traffic. The adaptation process also provides the user with the estimates of system detection effectiveness against the threats defined in the threat model, as it presents the effectiveness values for the currently selected aggregation function.

The background traffic is one of the adaptation process indirect inputs, as it influences the performance of the individual anomaly detection algorithms. As the network traffic is highly unpredictable, it is very difficult to predict which aggregation function will be chosen, especially given the fact that the challenges are selected from the DB using a stochastic process with a pseudo-random generator unknown to a potential attacker. The attacker therefore faces a dynamic IDS system that unpredictably switches its detection profile between several different profiles with utility (i.e. detection performance) values close to the optimum, and has to operate in a manner which would evade any of these profiles. This unpredictability, together with the additional robustness achieved by the use of multiple algorithms, makes the IDS evasion a much more difficult task than simply avoiding a single intrusion detection method[13].

The self-adaptation process (detailed in Fig. 2) is based on the insertion of challenges into the background of network flow data observed by the system. The challenges are represented as sets of NetFlow records, corresponding to classified incidents observed in the past. These records are generated by short lived, challenge specific *challenge agents* and are mixed with the background traffic, so that they cannot be distinguished from the background by the detection/aggregation agents. They are processed together with the rest of the traffic, used to update the anomaly detection mechanism data and trust models of individual detection agents and are evaluated with the rest of the traffic. Once the processing is completed, the challenge flows are re-identified by their respective challenge agents, removed from the user output and the anomaly attributed to these flows by individual aggregation agents is used to evaluate these agents and to select the optimal output agent for the current network conditions.

There are two broad types of challenges. The *malicious challenges* correspond to known attack types, while the *legitimate challenges* represent known instances of legitimate events that tend to be misclassified as anomalous. We further divide the malicious challenges into broad classes (denoted $AC_1, \ldots, AC_k, \ldots$) characterized by the type of the attack, such as fingerprinting/vertical scan, horizontal scan, password brute forcing, etc. These classes are used to make the connection between the threat models in Section 4.1 and the challenge selection. With respect to each of these attack classes, we characterize each aggregation agent by a probability distribution, empirically estimated from the continuous anomaly values attributed to the challenges from this class, as we can see in Fig. 3. We also define a single additional distribution for all legitimate challenges.

We assume that the anomaly values of both the legitimate and all types of malicious challenges define normal distributions, with the parameters $\bar{x}^k$ and σ_x^k for the k-th class AC_k of malicious challenges and $\bar{y}$ and σ_y for the legitimate ones[1]. The distance between the estimated mean values of both distributions ($\bar{x}^k$ and $\bar{y}$), normalized with respect to the values σ_x^k and σ_y represents the quality of the aggregation agent with respect to a given attack class. The *effectiveness* of the agent, defined as an ability to distinguish between the legitimate events and the attacks is defined as a weighted average of the effectiveness with respect to individual classes and will be estimated by the

[1] Normality of both distributions is not difficult to achieve, provided that the attack classes are properly defined and that the challenge samples in these classes are well selected, i.e. comparable in terms of size and other parameters.

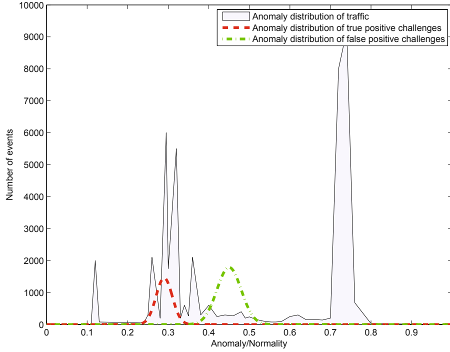


Fig. 3. Distribution of challenges on the background of the anomalies attributed to the traffic from one traffic observation interval. The distribution of anomaly of the malicious challenges (from one class) is on the left side of the graph, while the legitimate events are on the right.

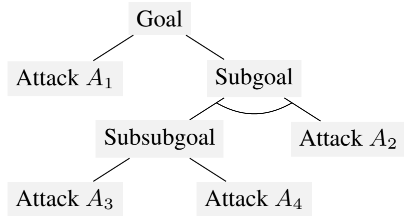


Fig. 4. Example Structure of an Attack Tree

trust modeling approach introduced in Sect. 5. In order to perform the above-described self-adaptation process, we need to address three important issues:

- offline selection of appropriate challenges and estimation of their relative importance (Sections 4 and 4.3),
- dynamic selection of the optimal aggregation agent to be used as a system output (Section 5), and
- dynamic determination of the optimal number of challenges.

4 Threat-Based Approach to Challenge Selection

In this section, we will present a method for challenge selection based on explicit threat modeling. We define a set $\mathcal{T} = \{T_1, \ldots, T_m\}$ of relevant *threats* as identified by the network administrator. Each threat is described by an attack tree, which specifies the adversary's attacks necessary to realize the threat. For each threat T_i, the system administrator has specified the expected damage $D(T_i)$, which would be caused should the attacker realize the threat. Our system uses challenges to evaluate its internal components in terms of accuracy and selects the most accurate component. Each challenge tests for a specific class of attacks. Therefore, the detection of threats can be directed by prioritizing those challenges that test for the most damaging threats.

In the following, we shortly review the concept of *attack trees* (Sect. 4.1) and show how they can be formulated in propositional calculus (Sect. 4.2). The latter allows us to minimize attack trees, and so bring them into an expedient form for further processing. We use the minimized attack trees to determine the composition of challenges for evaluating the internal components (Sect. 4.3).

4.1 Attack Trees

Attack trees depict how an attacker can attain a certain goal, e.g., to gain unauthorized access to a system resource. This overall goal constitutes a threat to a security system and builds the root of an attack tree.

The attack tree shows the alternative ways of how an attacker can reach the root, and so realize the threat. As formalized in [14], an attack tree is composed of AND and OR branches. Figure 4 shows a simple example of an attack tree structure. In this figure, the branch with a connectional arc depicts an *AND branch*, all other branches are *OR branches*. To reach the root, the attacker has to conduct a series of basic network attacks, e.g., "horizontal scan", which we call the *atomic attacks*. These atomic attacks constitute the leafs of an attack tree. An attacker "reaches" a leaf if he conducts the corresponding attack. Then, either if all children of a node with an AND branch are reached, then the node itself is reached. Similarly, an OR branched node is reached, if at least one of its children is reached. This way, starting at the leafs by conducting atomic attacks, an attacker can work its way up to the root. For our example in Fig. 4, the attacker can for instance reach the root by performing the attacks A_3 and A_2.

The principal advantages of the attack tree formalism are its simplicity, relatively high expressivity, and generality: an attack tree-level description of the threat is easily transferable between the networks and can be thus reused.

4.2 Attack Trees in Propositional Logic

We can say, an attacker can reach the root node by reaching specific subsets of the leafs. In this section we show how these specific subsets can be identified and minimized in a neat manner. First, we represent an attack tree in propositional logic. A formula corresponding to a tree should become true *iff* the main goal in the attack tree is attained. To build such a formula, we first create a literal for each atomic attack. Now, we successively go through the tree (starting from the root node), and connect all children of a node by the appropriate logic operation (OR for disjunctive branches, AND for conjunctive branches). Parentheses are used to group the children together. For the example tree shown in Fig. 4 this results in the formula:

$$(A_1) \vee ((A_3 \vee A_4) \wedge (A_2)) \ . \tag{1}$$

A formula is in Disjunctive Normal Form (DNF) iff it is a disjunction of conjunctive clauses. A formula is canonical, if all clauses contain all variables. We can bring any formula into canonical DNF by building a truth table that contains all variables, and taking all rows that evaluate to *true* as clauses. For our toy example in Fig. 4 that would result in:

$$(A_1 \wedge A_2 \wedge A_3 \wedge A_4) \tag{2}$$
$$\vee(A_1 \wedge A_2 \wedge A_3 \wedge \neg A_4) \tag{3}$$
$$\vee(A_1 \wedge A_2 \wedge \neg A_3 \wedge A_4) \tag{4}$$
$$\vee \qquad \ldots \tag{5}$$

Having an attack tree in canonical DNF, we can say, that an attacker realizes the threat if he succeeds to make at least one clause true. However, there is still much redundancy in the formula. For example, lines 2 and 3 together are logically equivalent to $A_1 \wedge A_2 \wedge A_3$. To remove all redundancy from the formula, we simply apply the Quine-McCluskey algorithm [15]. Note that when simplifying attack tree formulas, clauses will only contain positive literals. For the attack tree in Fig. 4, we finally get:

$$(A_1) \vee (A_3 \wedge A_2) \vee (A_4 \wedge A_2) \ . \tag{6}$$

A formula in DNF can be written as a set of clauses $\{C_1, C_2, \dots\}$ where each clause C_i is a set of positive literals $\{l_{i1}, l_{i2}, \dots\}$. We will write $F(T)$ for the minimal formula in DNF that corresponds to attack tree T. The attack tree from Fig. 4 can be formalized as:

$$F(T) = \{\{A_1\}, \{A_2, A_3\}, \{A_2, A_4\}\} \ . \tag{7}$$

4.3 Attack Tree Valuation

In this section, we first show how different attack classes can be prioritized, depending on the expected damage of the successful attacks, i.e. the attack tree root being attained by the adversary. We then show how the resulting priorities can be used to determine the composition of challenges for adapting the IDS. Finally, we exemplify the procedure with an example for a specific attack tree.

We assume, that a set of n detectable attacks $\mathcal{A} = \{A_1, \dots, A_n\}$ and general network conditions are known to the configured IDS. These attacks are classified into K attack classes $\{AC_1, \dots, AC_K\}$, with $\bigcup_k AC_k = \mathcal{A}$. We don't require that all attacks in an attack class are known, as the system is able to assess its effectiveness against the attacks inserted into the traffic in real-time. However, we require a sufficient set of attacks for each attack class, in order to use these samples as challenges.

The problem now is to prioritize the detection of attack classes. To this end, the following criteria should be fulfilled:

Attack trees: An attacker has a certain goal (which determines the attack tree T). Attack trees that cause more damage should be prioritized.

Clauses: An attacker tries to make one clause true in a chosen formula $F(T)$. Any clause made true causes the same damage $D(T)$. So each clause is assigned the same priority.

Literals: For making a chosen clause true, an attacker needs to make true *all* literals in this clause to cause damage $D(T)$. Therefore, all literals belonging to the same clause should be equally prioritized.

To fulfill the last two criteria, we compute the priority of an attack A_i within a tree T_j as follows:

$$P(A_i, T_j) := \frac{1}{|F(T_j)|} \sum_{\substack{C_k \in F(T_j), \\ \text{with } A_i \in C_k}} \frac{1}{|C_k|} \ . \tag{8}$$

The reader can easily verify that if A_i is not in T_j, then its priority within the tree is zero. Also, the sum of all priorities of the attacks in the tree is 1. To fulfill the first criterion, we additionally weight each tree T_j according to the damage $D(T_j)$ and get the final priority for an attack A_i by summing over all attack trees:

$$P(A_i) := \frac{1}{\sum_{T_j \in \mathcal{T}} D(T_j)} \cdot \sum_{T_k \in \mathcal{T}} D(T_k) \cdot P(A_i, T_k) \; . \tag{9}$$

Because of the normalization, again the priorities of all attacks sum up to 1. Hence, we can use these priorities to directly determine the ratio of challenges to test the respective attacks.

Procedure. In order to calculate the priorities of the attacks in $\mathcal{A}$, we propose the following procedure:

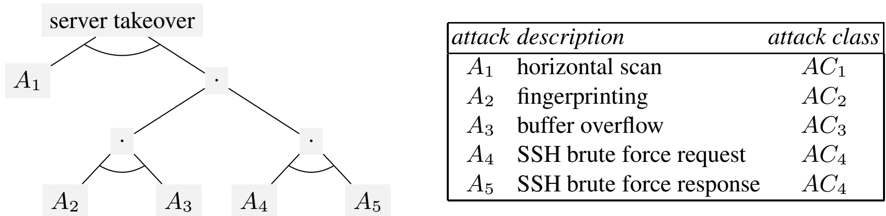
1. For each tree $T_i \in \mathcal{T}$ do:
   (a) Prune all impossible and non-detectable attacks from the tree.
   (b) Build $F(T_i)$: Transform the tree into a logical formula, bring it into DNF and minimize it (as in Sect. 4.2).
2. Compute $P(A_i)$ for each attack A_i as shown in formula (9).
3. For each attack class AC, add the priorities for all attacks in that class:

$$P(AC) = \sum_{A_i \in AC} P(A_i) \; . \tag{10}$$

   The ratio $P(AC)$ is a proportion of challenges from the class AC, and we will use it to as a weight in Eq. 22.

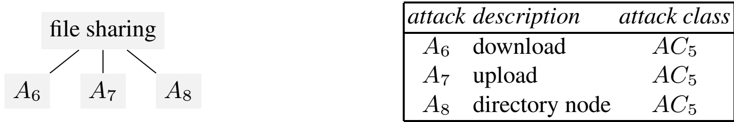
Example. In this section we show how the priorities are computed for a set of two very simple example attack trees T_1 and T_2 shown in Fig. 5 and 6 respectively. We estimate the damages of the trees to be $D(T_1) = 900$ and $D(T_2) = 100$. The minimal formulas in DNF for the two attack trees are:

$$F(T_1) = \{\{A_1, A_2, A_3\}, \{A_1, A_4, A_5\}\} \; , \tag{11}$$
$$F(T_2) = \{\{A_6\}, \{A_7\}, \{A_8\}\} \; . \tag{12}$$



	attack description	*attack class*
A_1	horizontal scan	AC_1
A_2	fingerprinting	AC_2
A_3	buffer overflow	AC_3
A_4	SSH brute force request	AC_4
A_5	SSH brute force response	AC_4

Fig. 5. Example Attack Tree T_1



attack	*description*	*attack class*
A_6	download	AC_5
A_7	upload	AC_5
A_8	directory node	AC_5

Fig. 6. Example Attack Tree T_2

We can now compute $P(A_i, T_j)$ for all attacks. Clearly $P(A_1, T_2) = 0$, so let us look at $P(A_1, T_1)$:

$$P(A_1, T_1) = \frac{1}{|F(T_1)|}\left(\frac{1}{|C_1|} + \frac{1}{|C_2|}\right) = \frac{1}{2}\left(\frac{1}{3} + \frac{1}{3}\right) = \frac{1}{3}\,. \tag{13}$$

Analogously we obtain:

$$P(A_2, T_1) = P(A_3, T_1) = P(A_4, T_1) = P(A_5, T_1) = \frac{1}{2} * \frac{1}{3} = \frac{1}{6}\,. \tag{14}$$

For attack tree T_2 we get:

$$P(A_6, T_2) = P(A_7, T_2) = P(A_8, T_2) = \frac{1}{3}\,. \tag{15}$$

Now, combining the two trees according to their expected damage, we obtain:

$$P(A_1) = \frac{D(T_1)}{D(T_1) + D(T_2)} \cdot P(A_1, T_1) = \frac{9}{10} \cdot \frac{1}{3} = \frac{3}{10}\,. \tag{16}$$

In the same way, we obtain for the other attacks:

$$P(A_2) = P(A_3) = P(A_4) = P(A_5) = \frac{3}{20}\,, P(A_6) = P(A_7) = P(A_8) = \frac{1}{30}\,. \tag{17}$$

Finally, we can compute the attack class priorities:

$$P(AC_1) = \frac{3}{10},\ P(AC_2) = P(AC_3) = \frac{3}{20}\,, P(AC_4) = \frac{3}{10},\ P(AC_5) = \frac{1}{10}\,. \tag{18}$$

5 Dynamic Aggregation Agent Selection

The insertion of challenges into the real traffic is not only a difficult problem from the technical perspective (due to the high volume of events processed in near-real-time and hard performance limitations of the system), but can also influence the effectiveness of the aggregation agents based on anomaly detection approaches. As these agents are not able to distinguish the challenges from the real events, the challenges are included in their traffic model, making it less representative of the background traffic and therefore reducing its predictive ability.

In this section, we present a trust-based algorithm which dynamically determines *the best aggregation agent* and also the *optimal number of challenges necessary for the reliable identification of the best aggregation agent*, while taking into account the: (*i*) past effectiveness of the individual aggregation agents and (*ii*) number of aggregation agents and the perceived differences in their effectiveness. We decided to use a trust-based approach for evaluating the aggregation agents, because it not only eliminates the noise in the background traffic and randomness of the challenge selection process, but accounts for the fact that attackers might try to manipulate the system by inserting misleading traffic flows. An attacker could insert fabricated flows [13] hoping they would cause the system to select an aggregation agent that is less sensitive to the threat the attacker actually intends to realize. When using trust, one tries to avoid this manipulation by dynamically adapting to more recent actions of an attacker.

For each time step $i \in \mathbb{N}$, the algorithm proceeds as follows:

1. Let each aggregation agent classify a set of challenges from different attack classes and selected legitimate challenges.
2. Update the *trust value* of each aggregation agent, based on its performance on the challenges in time step i.
3. Accept the output of the aggregation agent with the highest trust value as classification of the remaining events of time step i.

As we have stated above, we challenge detection and aggregation agents in each time step i with the sets of flows for which we already know the actual class, i.e. whether they are malicious or legitimate. So, we challenge an aggregation agent α with a set of malicious events, belonging to K attack classes and a set of legitimate events drawn from a single class. With respect to each class of attacks k, the performance of the agent is described by a mean and a standard deviation: $(\bar{x}^k, \sigma_x^k)$ for the set of malicious challenges and $(\bar{y}, \sigma_y)$ for the set of legitimate challenges. Both means lie in the interval $[0, 1]$, and $\bar{x}^k$ close to 0 and $\bar{y}$ close to 1 signify accurate classifications of the agent respectively (see Fig. 7). Based on this performance in time step i, we define the trust

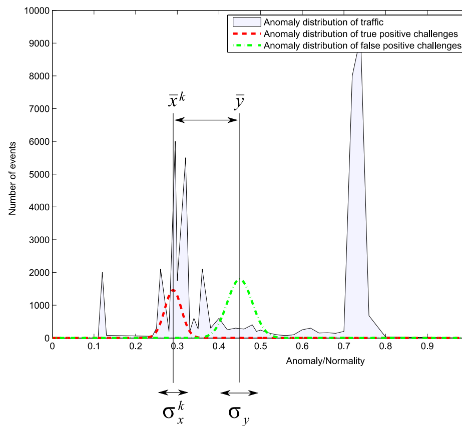


Fig. 7. Performance measures used for computing one trust experience

experience $t_\alpha^{i,k}$ with that aggregation agent α as follows:

$$t_\alpha^{i,k} = \frac{\bar{y} - \bar{x}^k}{\sigma_y + \sigma_x^k} \ . \tag{19}$$

The intention behind this formula is that an agent is more trustworthy, if its classifications are more *accurate* ($\bar{x}^k$ is low and $\bar{y}$ is high), and more *precise* (the standard deviations are low). Note that $t_\alpha^{i,k}$ lies in $(-\infty, \infty)$; however $t_\alpha^{i,k}$ will rarely be negative in practice.

To get the attack-class specific trust value T_α^k for an agent α, we aggregate the past trust experiences with that agent regarding the challenges from class k:

$$T_\alpha^k = \sum_i w_i * t_\alpha^{i,k} \ , \tag{20}$$

where w_i are weights that allow recent experiences a higher impact. This is done because older experiences are expected to be less significant than more recent ones. In our current system, the weights decrease exponentially. The system receives the input events in 5 minute batches, and assigns the same weight to all events in each batch. The weight of the challenges from the batch i is determined as:

$$w_i = \frac{1}{W} e^{(j-i)\frac{\ln(0.1)}{4}} \ , \tag{21}$$

where the j denotes the current time step, and the value of the coefficient $\frac{\ln(0.1)}{4}$ was selected so that challenges from the fifth batch (the oldest one being used) are assigned a weight of 0.1 before the normalization. The normalization is performed simply by dividing all weights by the sum of their un-normalized values W to ensure that $\sum w_i = 1$. We are currently using the challenges from the last 5 batches, meaning that the $(j - i)$ part of the exponent takes the values between 0 and 4. Please note that the specific assignment of weights w_i is highly domain specific, and is only included as an illustration of the general principle.

The final trust value T_α^i for the aggregation agent α is determined as a linear combination of the partial, attack class-specific values T_α^k:

$$T_\alpha^i = \sum_{k=1}^{K} P(AC_k) \cdot T_\alpha^k \ , \tag{22}$$

where the weights $P(AC_k)$ attributed to the trustworthiness of the individual classes are derived from Eq. 10.

5.1 Optimizing Number of Challenges

The number of challenges used as basis for the computation of the trust experiences $t_\alpha^{i,k}$ should be as small as possible while at the same time providing accurate results for the trust experiences. This means that we want to know the minimum number of challenges n for computing $\bar{x}^k$ and $\bar{y}$ which gives certain guarantees about the estimation of the actual means μ_{x^k} and μ_y (estimated by $\bar{x}^k$ and $\bar{y}$ respectively).

Guaranteeing margin of error m. At the outset, let us make two reasonable assumptions. First, we assume that the samples are normally distributed. This is the common assumption if nothing is known about the actual underlying probability distribution. Second, as suggested in [16], we assume the sample standard deviations which we found in past observations to be the actual standard deviations σ_x^k and σ_y. Then, the following formula gives us the number of challenges n that guarantees a specified margin of error m when estimating μ_{x^k} (or μ_y analogously) [16]:

$$n = \left(\frac{z^* \sigma_x^k}{m} \right)^2 , \tag{23}$$

where the critical value z^* is a constant that determines how confident we can be. Common critical values z^* are 1.645 for 90%, 1.960 for 95% and 2.576 for 99%. More specifically, the integral of the standard normal distribution in the range $[-z^*, z^*]$ equals the respective confidence level. If z^* is for instance chosen for a confidence level of 99%, we know that if we use n challenges for computing $\bar{x}^k$, the actual mean μ_{x^k} will lie in the interval $\bar{x}^k \pm m$ with the probability of 0.99.

Choosing margin of error m. The margin of error m is chosen such that we can be confident that the order of the first two most trustworthy agents is confirmed. In turn, this confirms that the selection of the first agent is the best choice. Let us call the first and the second agent α_1 and α_2 respectively, so we have $T_{\alpha_1} \geq T_{\alpha_2}$. We want to make sure that for the next trust experience this order is not reversed by chance. Recall that a trust experience t_α is defined as the difference between $\bar{y}$ and $\bar{x}^k$ weighted by the sum of the corresponding standard deviations (see formula (22)). As we use $2 * n$ challenges to find $\bar{y}$ and $\bar{x}^k$ respectively, the overall margin of error for the difference of $\bar{y}$ and $\bar{x}^k$ will not be higher than $2 * m$. The largest margin of error m' for which $t_{\alpha_1} \geq t_{\alpha_2}$ is still true (with the given confidence), must therefore fulfill the equation where t_{α_1} takes the lowest and t_{α_2} the highest possible value.

$$t_{\alpha_1} \geq \frac{\bar{y}_1 - \bar{x}_1^k - 2m'}{\underbrace{\sigma_{y_1} + \sigma_{x_1}{}^k}_{=:a}} = \frac{\bar{y}_2 - \bar{x}_2^k + 2m'}{\underbrace{\sigma_{y_2} + \sigma_{x_2}{}^k}_{=:b}} \geq t_{\alpha_2} , \tag{24}$$

where the inner equation can be solved to give:

$$m' = \frac{(t_{\alpha_1} - t_{\alpha_2})ab}{2(a+b)} = \frac{b(\bar{y}_1 - \bar{x}_1^k) - a(\bar{y}_2 - \bar{x}_2^k)}{2(a+b)} . \tag{25}$$

So, a choice of m with the constraint $m \leq m'$, guarantees with the specified confidence that we will get $t_{\alpha_1} \geq t_{\alpha_2}$ — in the case that this is the true order. To limit the number of challenges, we choose the maximal margin of error m that fulfills this constraint, which is given by $m := m'$. We also impose an additional lower bound on m, in order to prevent the number of challenges to grow disproportionally when the differences between the agent's trustworthiness with respect to this specific attack class AC_k are insignificant.

6 Experimental Evaluation

In the experimental part of our work, we evaluate two aspects of the mechanism: its ability to effectively reduce the number of false positives, while relying on an acceptable number of challenges, and its ability to selectively identify the events relevant to the priority threats as specified by the system administrator.

All the experiments were conducted on a university network, on the background of the regular network traffic. This background traffic contains roughly 10% of malicious flows, principally related to scanning, peer-to-peer activity, botnet propagation and brute force attacks on passwords, in no particular order.

In the first series of experiments, we test the ability of the suggested mechanism to produce the classifications with a reasonable error rate as expressed in terms of false positives and false negatives. To evaluate the error rate, we have manually classified the traffic from a significant subset of active hosts on the network. This classified traffic is then used to gauge the effectiveness of the method. The system observed about 80 000 flows every 5 minutes, with roughly 20 000 flows being malicious, and that the evaluation was performed over about seventy 5-minute long observation intervals. The system contained 30 aggregation agents, each of them averaging the opinions of the 5 underlying detection agents as described in Section 2.

In Fig. 8, we can see the number of challenges as it evolves over time. At the beginning, the system works with a fixed number of challenges, in order to let the anomaly detection methods in the detection agents adapt to the traffic. Once all the detection agents start (at step 5, after 25 minutes), the system starts to progressively insert more challenges, in order to build an initial assessment of all classifier agents. The number of challenges peaks at around the step 14, when it reaches 100 (all challenges combined). Once a user agent has built the initial trustworthiness for all agents, the number of challenges decreases until it levels out at around 40 (legitimate and malicious

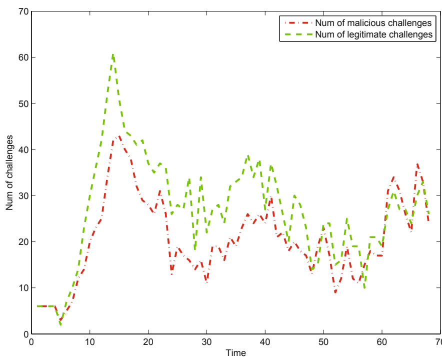


Fig. 8. Number of challenges over time, both legitimate (top, green curve) and malicious (bottom, red curve)

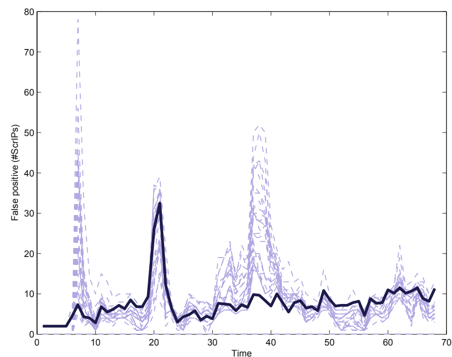


Fig. 9. Number of false positives (unique sources). Each aggregation agent is represented by one thin curve, the solid curve shows the performance of the aggregation agent dynamically selected by the system.

Table 1. Results of static system with arithmetic average (top line) compared to the selection of a single aggregation agent (middle part) and the dynamic self-adaptation mechanism described in this paper. Values are averaged to obtain the expected error numbers for one observation period.

Result	False Negat. [# sources]	False Posit. [# sources]
Arithmetic average	14.7	12.5
Average for aggregation fct.	13.1	24.3
Min FP for aggregation fct.	14.5	5.3
Min FN for aggregation fct.	9.8	125.2
Best aggregation fct.	13.7	5.7
Adaptive aggregation selection	14.0	3.1

challenges combined), where it fluctuates until the end of experiment. However, there are two notable increases to explain: between steps 30 and 40, and after step 60.

These increases can be easily explained when looking at Fig. 9, which shows the number of false positives in terms of unique source IP addresses. During these time intervals, we can notice that the choice of an appropriate aggregation agent has a huge impact on the quality of results, and that the adapted system is able to minimize the number of false positives. The number of challenges is lower between steps 40 and 60, when all agents provide similar results, and increases again around 60, where the performance of the aggregation agents varies somewhat more. On the other hand, we can see that the user agent did not manage to avoid a spike in false positives around the step 20, when it did not yet have a representative trust model.

The results shown in Fig. 9 are summarized in Table 1. We can see that the challenge-based, dynamic adaptation mechanism clearly outperforms the simple arithmetic average aggregation, which is the optimal selection when we have no information regarding the detection agent's performance. It also outperforms any single aggregation function selected using the **a-posteriori** knowledge from the pool of all 30 functions. All the methods have a comparable rate of false negatives, but differ in the rate of false positives, where the dynamic selection clearly outperforms the best aggregation functions. The relatively important margin of separation between the dynamic selection and best false positives of any single aggregation is given by the fact that the dynamic selection can avoid relatively high number of false positives during the periods when the individual aggregation functions differ in performance, such as around the sets 30-40. This further underlines the importance of the adaptive rate of challenge insertion, which allows fast identification of the optimal system output during the changes of system characteristics.

In Fig. 10, we can see the dynamics of the aggregation operator/agent selection over time. With an exception of the initial 6 intervals, when the operator #0 (arithmetic average) is selected by default, the system dynamically selects between the remaining operators, with about half of the selections being the operators #23 and #24. Both these operators include OWA as well as anomaly-detection-method-based weight average portion. They are identical in the fixed part, where they attribute the weight 0.33 to each of the agents Xu [7], MINDS [10] and TAPS [11]. The operators differ in the OWA part, where the first one builds its opinion from the three lowest anomaly values,

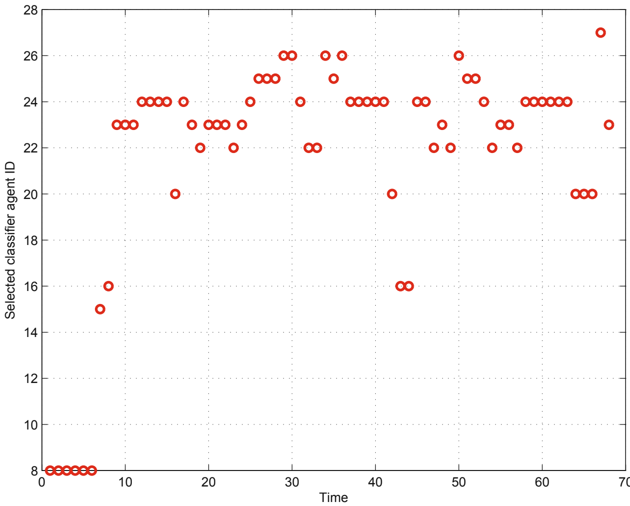


Fig. 10. Selected aggregation agent (identified by the ID number on axis y) for each time step

while the second considers the third and fourth anomaly values. The weights of the detection-method-based averaging and order weighted averaging parts are 0.5 for both operators. It is interesting to note that the system managed to pick the three methods with the most diverse set of anomaly detection features (in the fixed part), consistently with basic ensemble classification [17] principles. The quality of this result is therefore based not only on the absolute quality of underlying detection methods, but also benefits from the diversity of the anomaly detection methods.

In the above-described experiments, the challenges were inserted uniformly, regardless of the attack type. In the following, we will try to measure the effects of challenge insertion in terms of system sensitivity with respect to specific attacks. To do so, we have used the simple server compromise attack tree specified in Fig. 5 to generate the challenges optimizing the system, and we have then attempted to compromise one of the hosts on our network using the standard security tools, such as `nmap` or `metasploit`. The attacks were repeated several times, with changes in speed, tools settings and intensity. We have observed that the system selected the aggregation functions that were able to maximize the likelihood of detection of various stages involved in the server exploit attacks. The anomaly values attributed to horizontal sans, fingerprinting and vertical scans have increased considerably, making them far more likely to be detected. The most dramatic change of behavior was related to the password brute force breaking attempts. These were undetectable with the baseline system configuration, but became detectable with the case-specific system configuration. Buffer overflow attacks were undetectable regardless of the aggregation function, as they are nearly impossible to detect with NBA methods due to the low volumes of traffic involved.

In Table 2, we present the effects of threat model-based adaptation in the traffic used in the first series of experiments. This data set does not match the model at all and provides a good worst case example. We can see that the number of alerts (typically

Table 2. Effects of scenario specific selection on alert numbers in unrelated traffic. Obtained over 72 observation intervals 5 minutes long.

Result [# alerts]	False Negatives	False Positives	True Positives
Neutral challenge insertion	39	201	146
Case-specific insertion	37	249	161

greater than the number of malicious sources used in Table 1) generated by the system has grown, and that the number of false positives increased by about 50. The number of alerts classified as true positives have increased as well (by 15), and the number of false negatives decreased by 2. Note that the total number of alerts is not necessarily identical due to the possible alert fragmentation. Overall, we can see that in order to detect the attacks crucial in the server compromise scenario (e.g. password bruteforcing), the system was able to increase its sensitivity and to find a new equilibrium with different detection profile. It shall be also noted that most of the false positives are repetitive occurrences of traffic structures that are difficult to predict, and that about 80% of them can be eliminated with less than 20 rules in the alert processing engine.

7 Related Work

In literature, more sophisticated formalisms than attack trees have been proposed for modeling attack structures, e.g., attack graphs [18] and attack grammars [19]. However, for our purposes, we do not need to account for the order in which plans of attacks are carried out or the relations between attacks, and hence, the attack tree formalism is sufficiently rich.

In desktop grid computing, spot-checking [20,21] is used to make sure that hosts to which a computation has been outsourced, return correct results. To this end, indistinguishable challenges for which the correct answer is already known are interspersed with actual requests. For a spot-checking approach, where challenges are merged into a vector among a set of real requests, Staab et al. [2] showed how to determine an optimal number of challenges for a given number of real requests. They focused on the case where the answer to a challenge or a real request is binary. This was extended in our work, where we handle the continuous case.

The use of ensemble classification approaches [22] is functionally equivalent to our approach, but with extremely strong assumptions. It requires a pre-classified training data set and don't dynamically adapt system to the changing conditions.

Ghanbari and Amza [23] train belief networks that represent complex systems by injecting failures. At the outset, experts model a belief network that describes the dependencies within a system. The inserted failures then change the prior beliefs of the experts to form better estimates. Through fault injection, the dependencies between the variables in the belief network become evident, and so the overall system can be trained. Opposed to that, we inject challenges to evaluate classification components in terms of accuracy in order to select the most accurate one.

8 Conclusion

Our work presented in this paper aims to close the gap between security policies and formal threat models and the practice of IDS deployment. To achieve this objective, we have designed a runtime adaptation and monitoring framework running on the top of the IDS. It evaluates the performance with respect to the threat models, that are defined as attack trees, with a value assigned to the achievement the objective (root) of the each tree. Objective value can be defined in two manners. In a decision theoretical paradigm, we will aim to minimize our loss by associating an estimate of our loss (or risk) with the achievement of each attack tree root. In the game theoretic model, the value of the attack tree would reflect its value for the attacker. This second option allows us to differentiate between different types of attackers, with different technical capabilities represented by trees with growing complexity and corresponding risk values.

Either type of the threat/risk model can be used as an input for the online monitoring and adaptation process, which is able to evaluate the probability that an attack as defined by the attack tree would pass undetected. This results in an estimate of the **expected undetected loss**, given the current traffic status. This value is also a basis for system adaptation, as the system dynamically reconfigures itself in order to minimize the undetected loss value. The adaptation is based on the evaluation of system response with respect to a set of challenges, pre-classified recorded samples of the past traffic modified to fit the current traffic. The adaptation components of the system use the threat model to define the optimal mix of challenges to insert, in order to align the system performance with the threat models. It is also able to dynamically adjust the number of challenges to insert in response to changing traffic characteristics. The experiments performed with the system show that the dynamic selection of the optimal aggregation function in the CAMNEP system can significantly reduce the number of false positives and that the targeted insertion of challenges selected according to threat models can influence the system sensitivity to reflect the risks associated with each attack type.

The principal limitations of the work are related to the detection capabilities of the individual detection agents aggregated in the system. Using the assumption of classifier diversity [24], we know that the statistical performance of the combined classifier can be significantly better than the performance of individual classifiers. However, the system can not detect (i.e. separate from the traffic) the attacks that none of the individual algorithms can robustly detect.

In our future work, we plan to improve the attack modeling capabilities by inclusion plan-based attack modeling, and to integrate the outputs of the adaptation layer with the alert fusion and correlation capabilities of the system. This combination assess which attack stages are unlikely to be detected, and can use this information to improve the alert correlation [25].

Acknowledgment. This material is based upon work supported by the ITC-A of the US Army under Contract No. W911NF-08-1-0250. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ITC-A of the US Army. Also supported by Czech Ministry of Education grants 6840770038 (CTU) and 6383917201 (CESNET).

References

1. Denning, D.E.: An intrusion-detection model. IEEE Trans. Softw. Eng. 13, 222–232 (1987)
2. Staab, E., Fusenig, V., Engel, T.: Towards trust-based acquisition of unverifiable information. In: Klusch, M., Pěchouček, M., Polleres, A. (eds.) CIA 2008. LNCS (LNAI), vol. 5180, pp. 41–54. Springer, Heidelberg (2008)
3. Rehák, M., Pechoucek, M., Grill, M., Bartos, K.: Trust-based classifier combination for network anomaly detection. In: Klusch, M., Pěchouček, M., Polleres, A. (eds.) CIA 2008. LNCS (LNAI), vol. 5180, pp. 116–130. Springer, Heidelberg (2008)
4. Rehák, M., Pechoucek, M., Bartos, K., Grill, M., Celeda, P., Krmicek, V.: Improving anomaly detection error rate by collective trust modeling. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 398–399. Springer, Heidelberg (2008)
5. Cisco Systems: Cisco IOS NetFlow (2007), http://www.cisco.com/go/netflow
6. Scarfone, K., Mell, P.: Guide to intrusion detection and prevention systems (idps). Technical Report 800-94, NIST, US Dept. of Commerce (2007)
7. Xu, K., Zhang, Z.L., Bhattacharrya, S.: Reducing Unwanted Traffic in a Backbone Network. In: USENIX Workshop on Steps to Reduce Unwanted Traffic in the Internet (SRUTI), Boston, MA (2005)
8. Lakhina, A., Crovella, M., Diot, C.: Mining Anomalies using Traffic Feature Distributions. In: ACM SIGCOMM, Philadelphia, PA, pp. 217–228. ACM Press, New York (2005)
9. Lakhina, A., Crovella, M., Diot, C.: Diagnosis Network-Wide Traffic Anomalies. In: ACM SIGCOMM 2004, pp. 219–230. ACM Press, New York (2004)
10. Ertoz, L., Eilertson, E., Lazarevic, A., Tan, P.N., Kumar, V., Srivastava, J., Dokas, P.: Minds - minnesota intrusion detection system. In: Next Generation Data Mining. MIT Press, Cambridge (2004)
11. Sridharan, A., Ye, T., Bhattacharyya, S.: Connectionless port scan detection on the backbone, Phoenix, AZ, USA (2006)
12. Yager, R.: On ordered weighted averaging aggregation operators in multicriteria decision making. IEEE Transactions on Systems, Man, and Cybernetics 18, 183–190 (1988)
13. Rubinstein, B.I.P., Nelson, B., Huang, L., Joseph, A.D., Lau, S.-h., Taft, N., Tygar, J.D.: Evading anomaly detection through variance injection attacks on PCA. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 394–395. Springer, Heidelberg (2008)
14. Moore, A.P., Ellison, R.J., Linger, R.C.: Attack modeling for information security and survivability. Technical Report CMU/SEI-2001-TN-001, CMU Software Engineering Institute (2001)
15. Quine, W.: A way to simplify truth functions. American Mathematical Monthly 62, 627–631 (1955)
16. Moore, D.S.: The Basic Practice of Statistics, 4th edn. W. H. Freeman & Co., New York (2007)
17. Polikar, R.: Esemble based systems in decision making. IEEE Circuits and Systems Mag. 6, 21–45 (2006)
18. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: SP 2002: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Washington, DC, USA, p. 273. IEEE Computer Society, Los Alamitos (2002)
19. Zhang, Y., Fan, X., Wang, Y., Xue, Z.: Attack grammar: A new approach to modeling and analyzing network attack sequences. In: Proc. of the Annual Computer Security Applications Conference (ACSAC 2008), pp. 215–224 (2008)
20. Sarmenta, L.F.G.: Sabotage-tolerance mechanisms for volunteer computing systems. In: CCGRID 2001: Proc. of the 1st Int. Symposium on Cluster Computing and the Grid, Washington, DC, USA, p. 337. IEEE Computer Society, Los Alamitos (2001)

21. Zhao, S., Lo, V., GauthierDickey, C.: Result verification and trust-based scheduling in peerto-peer grids. In: P2P 2005: Proc. of the 5th IEEE Int. Conf. on Peer-to-Peer Computing, Washington, DC, USA, pp. 31–38. IEEE Computer Society, Los Alamitos (2005)
22. Giacinto, G., Perdisci, R., Rio, M.D., Roli, F.: Intrusion detection in computer networks by a modular ensemble of one-class classifiers. Information Fusion 9, 69–82 (2008)
23. Ghanbari, S., Amza, C.: Semantic-driven model composition for accurate anomaly diagnosis. In: ICAC 2008: Proceedings of the 2008 International Conference on Autonomic Computing, Washington, DC, USA, pp. 35–44. IEEE Computer Society, Los Alamitos (2008)
24. Dietterich, T.G.: Ensemble methods in machine learning. In: Kittler, J., Roli, F. (eds.) MCS 2000. LNCS, vol. 1857, pp. 1–15. Springer, Heidelberg (2000)
25. Morin, B., Mé, L., Debar, H., Ducassé, M.: M2D2: A formal data model for IDS alert correlation. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 115–137. Springer, Heidelberg (2002)

Malware Behavioral Detection by Attribute-Automata Using Abstraction from Platform and Language

Grégoire Jacob[1,2,⋆], Hervé Debar[1,⋆], and Eric Filiol[2]

[1] France Télécom R&D, Caen, France
{gregoire.jacob,herve.debar}@orange-ftgroup.com
[2] Operational Virology and Cryptology Lab., ESIEA Laval, France
filiol@esiea.fr

Abstract. Most behavioral detectors of malware remain specific to a given language and platform, mostly executables for Windows. The objective of this paper is to define a generic approach for behavioral detection based on two layers respectively responsible for abstraction and detection. The abstraction layer is specific to a platform and a language. It interprets the collected instructions, API calls and arguments and classifies these operations, as well as the objects involved, according to their purpose in the malware lifecycle. The detection layer remains generic and interoperable with different abstraction components. It relies on parallel automata parsing attribute-grammars where semantic rules are used for object typing (object classification) and object binding (data-flow). Theoretical results are first given with respect to the grammatical constraints weighting on the signature construction as well as to the resulting complexity of the detection. For experimentation purposes, two abstraction components have then been developed: one processing system call traces and the other processing the VBScript interpreted language. Experimentations have provided promising detection rates, in particular for scripts (89%), with almost no false positives. In the case of process traces, the detection rate remains significant (51%) but could be increased by sophisticated collection tools.

Keywords: Malware, Behaviors, Attribute-Grammars, Interpretation.

1 Introduction

Malware behavioral detection should theoretically be able to detect, if not innovative malware, at least unknown malware reusing variations of known techniques. However, most current behavioral detectors rely on specific characteristics, allowing evasion through simple functional modifications. This article aims to provide generic grammars modeling malicious behaviors in order

⋆ This work has been partially supported by the European Commissions through project FP7-ICT-216026-WOMBAT funded by the 7th framework program. The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 81–100, 2009.
© Springer-Verlag Berlin Heidelberg 2009

to build efficient and resilient detection automata. Deterministic finite automata are attractive because their linear complexity remains acceptable for operational deployment. In 1995, [1] already used automata to describe the alternative sequences of operations making up malicious behaviors. Since then, researches focusing on the notion of data flow has led to the apparition of tainting techniques to detect malicious uses of data [2]. Control of the data flow has exhibited significant successes and is now broadly used, in intrusion detection [3] or malware behavior extraction [4]. These articles use automata to model the sequences of system calls constituting respectively attacks and behaviors. The data flow is then captured by analysis of the parameters collected along the calls. On this principle, [5] focuses on self-reproduction as the discriminating behavior for detection.

Similarly, our approach of behavioral detection combines automata and data flow control. The model easily supports multiple behaviors. In fact, malicious behaviors are described by attribute-grammars. Syntactic rules describe the possible combinations of operations making up the behavior, whereas, semantic rules both control the data flow between the involved objects, and associate them with a potential purpose in the malware lifecycle (installation, communication, execution). The detection process is finally achieved by parsing execution traces to check for the satisfaction of the grammatical behavior descriptions.

Abstraction is needed to translate observed traces into the behavioral model for detection. By a layered architecture, [6] addresses the semantic gap existing between the system call traces, understandable by OS specialists, and high-level behaviors. Similarly, our abstraction layer provides generic descriptions where the processed data get detached from the specificities of the platform and the programming language. In fact, the graph-based formalism in [6] is in many ways equivalent to the grammatical formalism provided here. In effect, AND/OR graphs may be expressed by the semantic rules of attribute-grammars. Relying on a well-established formalism, these grammars provide theoretical results in terms of complexity which also hold for the approach from [6]. In addition, the present article provides different behaviors, assessed on larger test pools.

With regards to the operations for language abstraction, the identification of the system objects with a potential use for malware and the generation of the grammatical behavior descriptions, they all require an initial configuration step as described in Fig.1. Contrary to other methods, the configuration focuses both on critical objects, which remain enumerable in a standard environment, and innovative malware, which are scarce among the numerous variants of known malware. In a few words, this paper introduces the following contributions:

- A model of malicious behaviors using attribute-grammars with semantic for object binding (data flow control) and typing (object purposes for malware).
- An abstraction layer to translate observed traces into the model, detaching detection from the specificities of platforms and programming languages, with two proofs of concept to analyze executable traces and scripts.
- Some generic automata for behavior detection with an assessment from perspectives theoretical (complexity) and operational (coverage, performance).

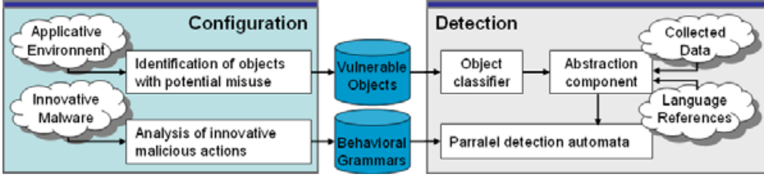


Fig. 1. Configuration and detection processes

The article is articulated as follows. Section 2 introduces the behavioral model based on attribute-grammars. Section 3 presents the abstraction process from the collected data to the model. Section 4 describes the detection process. An implementation is given in Section 5 whose results are commented in Section 6.

2 Grammatical Formalization of Behaviors

From a theoretical perspective, an attribute-grammar (Definition 1) is a Context-Free Grammar (CFG) enriched with semantic attributes and rules [7]. In the formalism, each start symbols begins the description of a new malicious behavior. The terminal symbols of the grammar then correspond to the basic operations making up the behavior whereas the production rules describe their different combinations to achieve the behavior. As stated in [8,9], basic operations eventually refer to data collected through the abstraction layer (instructions, API calls, parameters). These common principles are kept along the formalization.

Definition 1. *An attribute-grammar* G_A *is a triplet* $<G, D, E>$ *where:*
- G is originally a context-free grammar $<V, \Sigma, S, P>$*,*
- $att : X \in \{V \cup \Sigma\} \rightarrow att(X) \in Att^*$ *is an assignment function for attributes and* $D = \cup_{\alpha \in Att} D_\alpha$ *their set of values,*
- E is a set of semantic rules such as for any production of P, there is at most one rule per variable of the form $Y.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$ *with* $f : D_{\alpha_1} \times ... \times D_{\alpha_n} \rightarrow D_\alpha$*.*

2.1 Malicious Behavior Language

A generic programming language is required to describe malicious behaviors: the Malicious Behavior Language (MBL) has been designed to this purpose. Its syntax and operational semantics are given in [8]. Most malicious behaviors can be described by sub-grammars of the MBL generative grammar. The language principles are object-oriented according to the encapsulation in Fig.2. It provides internal operations: arithmetic and control operations guaranteeing Turing completeness, as well as interactions to interface with external objects: commands (open, create, close, delete, execute) or inputs/outputs (send, receive).

On top of the syntax for operations and interactions, a type system has been provided for the external objects. These objects are typed according to

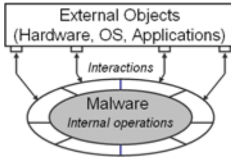


Fig. 2. Malware object encapsulation

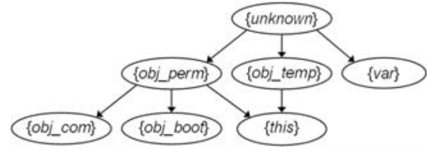


Fig. 3. Object type poset

their potential use in the malware lifecycle: permanent objects (*obj_perm*), temporary objects (*obj_temp*), booting objects (*obj_boot*), communicating objects (*obj_com*), self-reference (*this*). A partial order has been defined on these types according to their subset inclusion, as shown in the Hasse Diagram of Fig.3. These inclusions correspond to object specializations. This type system can be deployed thanks to the semantic attributes enriching attribute-grammars. In fact, semantic attributes and rules can have several purposes:

Object binding: Object binding identifies the different object instances, and guarantees they are coherently used. It is achieved by affecting specific attributes called identifiers to the terminal symbols representing objects (denoted $objId$ where Id is an abbreviation for Identifier). Considering interactions, the binding constrains the data-flow between objects. The data flow is critical in behaviors such as duplication where data transfers are involved.

Object typing: A type attribute can also be affected to a given object (denoted $objTp$ where Tp is an abbreviation for Type). Types are attached to objects according to their potential use. They are critical to distinguish certain malicious purposes such as booting objects in the case of residency or communicating objects in the case of propagation. Additional characterization of the objects can be achieved through additional attributes. For example, an attribute can store the object nature (denoted $objNat$): variable, file, registry key, network socket, mail, etc.

2.2 Descriptions of Malicious Behaviors

Four behaviors are examined: duplication, propagation, residency (automatic start) and overinfection tests (avoiding reinfection of an infected system). Because their whole descriptions would be too tedious, only two extracts of the most prevalent ones are covered: duplication and propagation. Their descriptions, as well as additional behaviors, had been generated in [8], by manual analysis of a malware pool. Since these descriptions convey the most generic features of the malicious behaviors, manual generation can be considered more easily than for the binary signatures of scanners.

Duplication. Duplication is achieved by copying code from the self-reference to a permanent object. It is described below by syntactic production rules (grey) and their related semantic rules (white). The syntactic derivations correspond

to different duplication techniques: only single-block read/write is described above, but the complete description also supports interleaved read/write and direct copy. The semantic rules guarantee the data-flow through a same variable between read and write interactions (Binding: <*Write*>.varId = <*Read*>.varId). They also guarantee the behavior maliciousness by constraining read interactions to refer to the self-reference (Typing: <*Duplicate*>.srcTp = *this*).

(*i*) <*Duplicate*>	::=	<*Create*><*Open*> <*Read*><*Write*>
	|	<*Open*><*Create*> <*Read*><*Write*>
	|	<*Open*><*Read*> <*Create*><*Write*>
{ <*Duplicate*>.srcId	=	<*Open*>.objId
<*Duplicate*>.srcTp	=	*this*
<*Duplicate*>.targId	=	<*Create*>.objId
<*Duplicate*>.targTp	=	*obj_perm*
<*Open*>.objTp	=	<*Duplicate*>.srcTp
<*Create*>.objTp	=	<*Duplicate*>.targTp
<*Read*>.objId	=	<*Duplicate*>.srcId
<*Read*>.objTp	=	<*Duplicate*>.srcTp
<*Write*>.objId	=	<*Duplicate*>.targId
<*Write*>.objTp	=	<*Duplicate*>.targTp
<*Write*>.varId	=	<*Read*>.varId }

(*ii*) <*Create*>	::=	*create object*;
{ <*Create*>.objId	=	*object*.objId
object.objTp	=	<*Create*>.objTp }
(*iii*) <*Open*>	::=	*open object*;
{ <*Open*>.objId	=	*object*.objId
object.objTp	=	<*Open*>.objTp }
(*iv*) <*Read*>	::=	*receive object*1 ← *object*2;
{ <*Read*>.varId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*2.objId	=	<*Read*>.objId
*object*2.objTp	=	<*Read*>.objTp }
(*v*) <*Write*>	::=	*send object*1 → *object*2;
{ <*Write*>.varId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*2.objId	=	<*Write*>.objId
*object*2.objTp	=	<*Write*>.objTp }

Propagation. Propagation differs from duplication by a different target. The malware code is copied from the self-reference to a communicating object. Consequently, it shows syntactic similarities with duplication, except adjustments to insert a format process. The main differences thus lie in adaptations of the semantic rules. Illustrating typing, the permanent type of the target is replaced by the communicating type (<*Propagate*>.targTp = *obj_com*). A communicating object can either be a network connection, a mail or a shared file. The second modification specifies, by a disjunction of semantic equations, that the propagation source can be either the self-reference or the result of a previous duplication (<*Propagate*>.srcTp = *this* or <*Propagate*>.srcId = <*Duplicate*>.targId).

(*i*) <*Propagate*>	::=	<*Open*><*Read*><*Transmit*>
	|	<*Read*><*Open*><*Transmit*>
{ <*Propagate*>.srcId	=	<*Read*>.objId
(<*Propagate*>.srcTp = *this*	∨	<*Propagate*>.srcId = <*Duplication*>.targId)
<*Propagate*>.targId	=	<*Open*>.objId
<*Propagate*>.targTp	=	*obj_com*
...		}
(*ii*) <*Transmit*>	::=	<*Format*><*Write*> | <*Write*>

3 Model Translation by Abstraction

In the context of behavioral detection, a trace conveying the actions of the monitored program is statically or dynamically collected. Depending on the collection mechanism, completeness of the data and its nature vary greatly, from simple instructions to system calls along with their parameters. The trace remains specific to a given platform and to the language in which the program has been

coded (native, interpreted, macros). An abstraction layer is thus required for translation into the behavioral language from Section 2. Translation of basic instructions, either arithmetic (move, addition, subtraction...) or control related (conditional, jump...), into operations of the language is an obvious mapping, not requiring further explanation. On the opposite, translation of API calls and their parameters into interactions and objects from the language is detailed thereafter.

3.1 API Calls Translation

For a program to access services and resources, the Application Programming Interfaces (APIs) constitute a mandatory point enforcing security and consistency [10]. API calls are also denoted system calls when accessing services from the operating system. For each programming language, the set of available APIs can be classified into distinct interaction classes. This set being finite and supposedly stable, the translation is defined as a mapping over the interaction classes, the completeness of the process being guaranteed. Table 1 provides a mapping for APIs subsets from Windows [11] and VBScript. The table is refined according to the nature of the manipulated objects. The API name, on its own, is not always sufficient to determine its interaction class. For example, network devices and files use common APIs; the distinction is made on their path (`\device\Afd\Endpoint`). Sending, receiving packets then depends on control codes transmitted to `NtDeviceIoControlFile` (`IOCTL_AFD_RECV`, `IOCTL_AFD_SEND`). If required, specific call parameters constitute additional mapping inputs:

$\{\text{API name}\} \times (\{\text{Parameters}\} \cup \{\epsilon\}) \rightarrow \{\text{Interaction class}\}$.

3.2 Parameters Interpretation

In the context of interactions, parameters are important factors to identify the involved objects and assess their criticality through typing. Parameters interpretation thus complements the initial abstraction from the platform and language obtained through API translation. Due to their varying nature, parameters can

Table 1. Mapping Windows Native and VBScript APIs to interaction classes

Interaction Class	Object Nature	Windows Native API	VBScript API
Open	File	**NtOpenFile**(ptr FileHandle, ..., str FilePath, ...) **NtCreateSection**(ptr SectionHandle, ..., ptr FileHandle)	FileSystemObject.**GetFile** (str FilePath) FileSystemObject.**OpenTextFile**(str FilePath) FileSystemObject.**Drives.Item** (int DriveNb) ...
	Network	**NtOpenFile**(ptr DeviceHandle,...,str NetworkDevicePath,...)	
Read	File	**NtReadFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtMapViewOfSection**(ptr SectionHandle, ..., ptr BaseAddress,...) ...	FileObject.**Read**() FileObject.**ReadLine**() FileObject.**ReadAll**()
	Network	**NtDeviceIoContolFile**(ptr DeviceHandle, ...,ReadCtrl, ptr Buffer,...)	
Write	File	**NtWriteFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtWriteFileGather**(ptr FileHandle, ..., ptr SegmentArray, ...)	FileObject.**Write**(str Value) FileObject.**WriteLine**(str Value) FileObject.**Copy**(str FilePath) ...
	Network	**NtDeviceIoContolFile**(ptr DeviceHandle, ...,SendCtrl, ptr Buffer,...)	
	Mail		MailObject.**TextBody**(str Content) MailObject.**AddAttachment**(str FilePath) ...

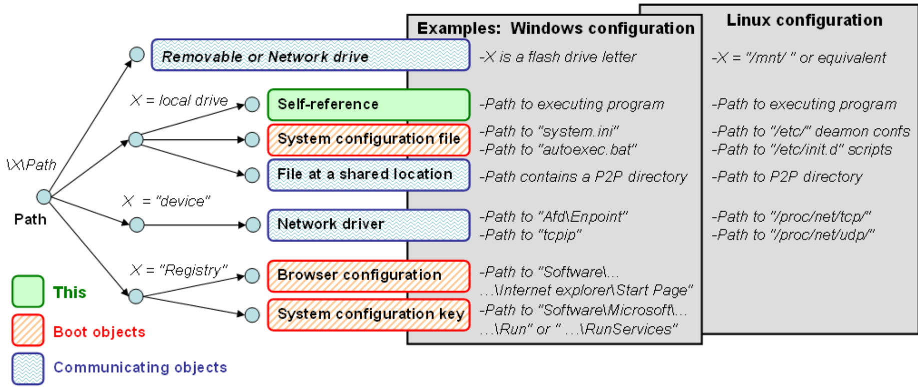


Fig. 4. Character strings interpretation

not be translated by a simple mapping. Decision trees are more adaptive tools, capable of interpreting parameters according to their representation:

Simple integers: Integer attributes are mainly constants specific to an associated API. They may condition the interpretation of its interaction class.

Address and Handles: Addresses and handles identify the different objects appearing in the collected traces. They are particularly useful to study the data flow between objects. Considering a variable, it is represented by its address a_v and its size s_v. Every address a such as $a_v \leq a \leq a_v + s_v$ will refer to the same variable. Certain addresses with important properties may be refined by typing: import tables, services table, entry points. These specific addresses may be interpreted by decision trees partitioning the address space.

Character strings: String parameters contain the richer information. Most of these parameters are paths satisfying a hierarchical structure where every element is important: from the root identifying drives, drivers and registry, passing by the intermediate directories providing object localization, until the real name of the object. This hierarchical structure is well adapted for a progressive analysis embedded in a decision tree. A progressive interpretation of the path elements is shown in Fig.4 with basic examples for Windows and Linux platforms.

3.3 Decision Trees Generation

Building decision trees requires a precise identification of the critical resources of a system. Our methodology proceeds by successive layers: hardware, operating system and applications. For each layer, we define a scope encompassing the significant components; the resources involved either in the installation, the configuration or the use of these components are monitored for potential misuse:

Hardware layer: For the hardware layer, the scope can be restricted to the interfaces open to external locations (Network, CD, USB). The key resources

to monitor are the drivers used to communicate with these interfaces as well as additional configuration files (e.g. `Autorun.inf` files impacting booting).

Operating system layer: OS configuration is critical but unfortunately dispersed in various locations (e.g. files, registry, structures in memory). However, most of the critical resources are already well identified, such as the boot sequence or the intermediate structures used to access the provided services and resources (e.g. file system, process table, system call table).

Applicative layer: It is obviously impossible to consider all existing applications. To restrict the scope, observing malware propagation and interoperability constraints, the analysis is limited to connected and widely deployed applications (web browsers, messaging, mail, peer-to-peer, IRC clients). Again are considered resources involved in communication (connections, transit locations) as well as in configuration (application launch).

Identification of the critical resources potentially used by malware is a manual, but necessary, configuration step. We believe however that it is less cumbersome than analyzing the thousands of malware discovered every day, for the following reasons. First, critical resources of a given platform are known and limited; they can thus be enumerated. Their name and location can then be retrieved in a partially automated way (e.g. listing connected drives, recovering peer-to-peer clients and their shared folders). In fact, full automation of the parameter interpretation may be hard to achieve. In [12], an attempt was made to fully automate their analysis for anomaly-based intrusion detection. The interpretation relied on deviations from a legitimate model based on string length, character distribution and structural inference. These factors are significant for intrusions which mostly use misformatted parameters to infiltrate through vulnerabilities. It may prove less efficient with malware since they can use legitimate parameters, at least in appearance. Moreover, the real purpose of these parameters would still be unexplained; an additional analysis would be required for type affectation. Thus, interpretation by decision trees with automated configuration seems a good trade off between automation and beforehand manual analysis.

4 Detection Using Parsing Automata

Detecting malicious behaviors may be reduced to parsing their grammatical descriptions. To achieve syntactic parsing and attribute evaluation in a single pass, the attribute-grammars must be both LL grammars and L-attribute grammars: attribute dependency is only allowed from left to right in the production rules. These properties are not necessarily satisfied by the MBL generative grammar but they prove true for the sub-grammars describing the malicious behaviors. Therefore, detection can be implemented by LL-parsers, capable of building, from top to down, the annotated leftmost-derivation trees. Basically, LL-parsers are pushdown automata enhanced with attribute evaluation (Definition 2).

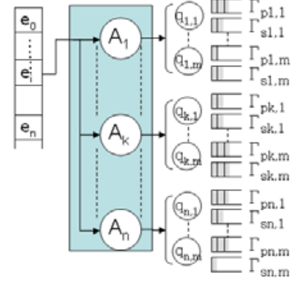
Definition 2. *A LL-parser is a particular pushdown automaton A that can be built as a ten-tuple* $<Q, \Sigma, D, \Gamma_p, \Gamma_s, \delta, q_0, Z_{p,0}, Z_{s,0}, F>$ *where:*

- Q is the finite set of states, and $F \subset Q$ is the subset of accepting states,
- Σ is the alphabet of input symbols and D is the set of values for attributes,
- Γ_p / Γ_s are the parsing / semantic stack alphabets,
- $q_0 \in Q$ is the initial state and $Z_{p,0}$ / $Z_{s,0}$ are the stacks start symbols,
- δ is the transition function defining the production rules and semantic routines, of the form: $Q \times (\{\Sigma \cup \epsilon\}, D^*) \times (\Gamma_p, \Gamma_s) \rightarrow Q \times (\{\Gamma_p \cup \epsilon\}, \Gamma_s)$.

Several behaviors are monitored in parallel by dedicated automata. Each automaton A_k parses several instances of the behavior, storing its progress in independent derivations (triple made up of a state q_k and parsing and semantic stacks Γ_{pk}, Γ_{sk}). For each collected events e_i containing input symbols and semantic values, all the parsing parallel automata progress along their derivations. When an irrelevant input is read (an operation interleaved inside the behavior for example), this input is dropped instead of causing an error state. The global procedure is defined in the Algorithms 1 and 2 with an explicative figure.

Algorithm 1. A.ll-parse(e,Q,Γ_p,Γ_s)

1: **if** e, Q, Γ_p, Γ_s match a transition $T \in \delta_A$ **then**
2: **if** e introduces a possible ambiguity **then**
3: duplicate state and stack triple (Q, Γ_p, Γ_s).
4: **end if**
5: Compute transition T to update (Q, Γ_p, Γ_s).
6: **if** Q is an accepting state $Q \in F_A$ **then**
7: Malicious behavior detected.
8: **else**
9: ignore e.
10: **end if**
11: **end if**



Algorithm 2. BehaviorDetection(e_1,...,e_t)

Require: events e_i are couples of symbol and semantic values: $(\{\Sigma \cup \epsilon\}, D^*)$.
1: **for all** collected events e_i **do**
2: **for all** the automata A_k such as $1 \leq k \leq n$ **do**
3: m_k = current number of parallel derivations handled by A_k.
4: **for all** state and stack triple $(Q_{k,j}, \Gamma_{pk,j}, \Gamma_{sk,j})$ such as $1 \leq j \leq m_k$ **do**
5: A_k.**ll-parse**$(e_i, Q_{k,j}, \Gamma_{pk,j}, \Gamma_{sk,j})$).
6: **end for**
7: **end for**
8: **end for**

4.1 Semantic Prerequisites and Consequences

The present detection method can be related to scenario recognition in intrusion detection. An intrusion scenario is defined as a sequence of dependent attacks [13]. For each attack to occur, a set of prerequisites or preconditions must be

satisfied. Once completed, new consequences are introduced, also called postconditions. In [14], isolated alerts are correlated into scenarios by parsing attribute-grammars annoted with semantic rules to guarantee the flow between related alerts. Similarly, a malicious behavior is a sequence where each operation prepares for the next one. In a formalization by attribute grammars, the sequence order is led by the syntax whereas prerequisites and consequences are led by semantic rules of the form $Y_i.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$ (Definition 1).

Checking prerequisites: Prerequisites are defined by specific semantic rules where the left-side attributes of the equations are attached to terminal symbols ($Y_i \in \Sigma$). During parsing, semantic values are collected along input symbols. These values are compared to values computed using inherited and already synthesized attributes. This comparison corresponds to the matching step performed on the semantic stack Γ_s during transitions from δ.

Evaluating consequences: When the left-side attribute is attached to a non-terminal ($Y_i \in V$) and right-side attributes are valued, the attribute is evaluated. During transitions from δ, the evaluation corresponds to the reduction step where the computed value is pushed on the semantic stack Γ_s.

4.2 Ambiguity Support

All events are fed to the behavior automata. However, some of them may be unrelated to the behavior or unuseful to its completion. Unrelated events do not match any transition and are simply dropped. This is insufficient for unuseful events raising ambiguities: they may be related to the behavior but parsing them makes the derivation fail unpredictably. Let us take an explicit example for duplication. After opening the self-reference, two files are consecutively created. If duplication is achieved between the self-reference and the first file, parsing succeeds. If duplication is achieved with the second one, parsing fails because the automaton has progressed beyond the state of accepting a second creation. Similar ambiguities may be observed along the variable affectations which alter the data-flow. The algorithm should thus be able to manage the different objects and variables combinations. Ambiguities are handled by the detection algorithm using derivation duplicates. Before transition reduction, if the operation is potentially ambiguous, the current derivation is copied in a new triple containing the current state and the parsing and semantic stacks. This solution handles the combinations of events without backtracking. To come back and forth in the derivation trees would have proved too cumbersome for real-time detection.

4.3 Time and Space Complexity

LL-parsing is linear in function of the number of symbols. Parallelism and ambiguities increase the complexity of the detection algorithm. Let us consider calls to the parsing procedure as the reference operation. This procedure is decomposed in three steps: matching, reduction and accept (two comparisons and a computation). In the worst case scenario, all events are related to the behavior

automata and all these events introduce ambiguities. In the best case scenario, no ambiguity is raised. Resulting complexities are given in Proposition 1.

Proposition 1. *In the worst case, behavioral detection using attributed automata has a time complexity in* $\vartheta(k(2^n - 1))$ *and a space complexity in* $\vartheta(k2^n(2s))$ *where* k *is the number of automata,* n *is the number of input symbol and* s *is the maximum stack size. In the best case, time complexity drops to linear time* $\vartheta(kn)$ *and space complexity becomes independent from the number of inputs* $\vartheta(k2s)$*.*

The worst case complexity is important but it quickly drops as the number of ambiguous events decreases. The experimentations in Section 6 show that the ratios of ambiguous events are limited and the algorithm offers satisfactory performances. Based on these ratios, a new assessment of the average practical complexity is provided. Besides, these experimentations also show that important ratios of ambiguous events are already a sign of malicious activity.

Proof. In a best case scenario, the number of derivation for each automaton remains constant. Considering the worst case scenario, all events are potentially ambiguous for all the current derivations. Technically, ambiguities multiply by two the number of derivations at each iteration of the main loop. Consequently, each automaton handles 2^{i-1} different derivations at the ith iteration. The time complexity is then equivalent to the number of calls to the parsing procedure:

(1) $k + 2k + ... + 2^{n-1}k = k(1 + 2 + ... + 2^{n-1}) = k(2^n - 1)$

The maximum number of derivations is reached after the last iteration where all automata manage 2^n parallel derivations. Each derivation is stored in two stacks of size s. This moment coincide with the maximum memory occupation:

(2) $k2^n(2s)$.

5 Prototype Implementation

The prototype includes the aforementioned two layers: a specific collection and abstraction layer and a generic detection layer. The overall architecture is described in Fig.5. Components of the abstraction layer interpret the specificities of the languages whereas the common object classifier interprets the specificities of the platform. As a proof of concept, abstraction components have been implemented for two languages: native code of PE executables and interpreted Visual Basic Script. Above abstraction, the detection layer based on parallel behavioral automata parses the interpreted traces independently from their original source.

5.1 Analyzer of Process Traces

Process traces provide useful information about the system activity of an executable. The detection method could be deployed in real-time but for a greater easiness, the experimentations were led off-line. The process traces were thus collected beforehand inside a virtual environment to avoid any risk of infection. The

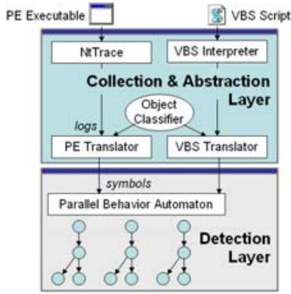


Fig. 5. Multi-layer architecture

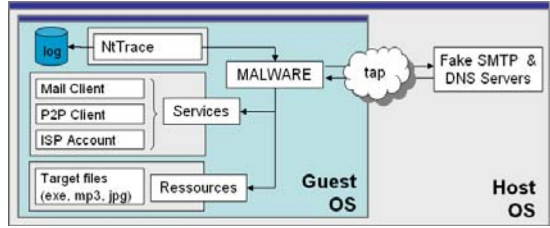


Fig. 6. Collection environment for API calls

prototype deploys an existing tool called NtTrace to collect Windows native calls, their arguments and returned values [15]. The main point with dynamic collection mechanisms (real-time or emulation based) is that most behaviors are conditioned by external objects and events: available target for infection or listening servers for network propagation. In order to increase the mechanism coverage and collect conditioned behaviors, the virtual environment from Fig.6 has been deployed over Qemu [16]. Windows XP was installed on the drive image and useful services and resources were configured: system time, ISP account, Mail and P2P clients, potential targets (.exe, .jpg, .html). Outside the virtual machine, emulations of DNS and SMTP servers have been deployed to establish connections and capture a network activity at the system call level. The weight of the platform and its configuration may seem important but notice that simple call interception would be sufficient in a real-time deployment without any need for a virtual environment.

Translation is then deployed line by line on the collected traces. It directly implements the results from Section 3 for API call translation and parameter interpretation. Only a selection of APIs is classified by mappings; the others are ignored until their future integration. An object classifier, embedding decision trees specifically crafted for a Windows configuration as in Fig.4, is then called on the parameters. During the process, sequences of identical or combined calls are detected and formatted into loops in order to compress the resulting logs. Looking specifically at creation and opening interactions, when resolved, a correspondence is established between the names of objects and their references (addresses, handles). Following interactions check for these references for interpretation. Conversely, on deletion or closing, this correspondence is destroyed for the remainder of the analysis. Names and identifiers must be unlinked since a same address or handle number could be reused for a different object.

5.2 Analyzer of Visual Basic Scripts

No collection tool similar to NtTrace was available for VBScript. We have thus developed our own collection tool, directly embedding the abstraction layer. VB-Script being an interpreted language, its static analysis is simpler than native

code because of the visibility of the source but also because of some integrated safety properties: no direct code rewriting during execution and no arbitrary transfer of the control flow [17]. For these reasons, path exploration becomes conceivable. The interest of the static approach with respect to the dynamic one used for process traces lies in the coverage of the collected data. In effect, the different potential actions corresponding to the different execution paths will be monitored. In addition, the visibility over the internal data flow will be increased likewise. By comparison, the results of the experimentations will eventually be a good indicator of the impact of the collection mechanism on detection.

Basically, the VBScript analyzer is a partial interpreter using static analysis for path exploration. The analyzer is divided into three parts:

1) Static analyzer: The static analyzer heavily depends on the syntactic specifications of the VBScript language [18]. The script is first parsed to localize the main, the local functions and procedures, as well as to retrieve their signature. Its structure is then parsed by blocks to recover information about the declared variables and instantiated managers (file system, shell, network, mail). In addition, the analyzer also deploys code normalization to remove the syntactic shortcuts provided by VBScript, but most critically to thwart obfuscation. By normalization, the current version can handle certain categories of obfuscation such as integer encoding, string splitting or string encryption.

2) Dynamic interpreter: A partial interpreter has been defined to explore the different execution paths. It is only partial in the sense that the script code is not really executed. Only significant operations and dependencies are collected. To support path exploration, the analyzer handles conditional and loop structures, but also calls to local functions and procedures. Inside these different blocks, each line is processed to retrieve the monitored API calls manipulating files, registry keys, network connections or mails. Calls interpretation is deployed by mapping as previously defined. Affectations, impacting the data-flow, are thereby also monitored. Additional analysis is then deployed to process the expressions used as call arguments, or affected values. In order to control the data-flow, object references and aliases must be followed up through the processing of expressions:
- Local function/procedure calls: linking signature with the passed parameters,
- Monitored API calls: creating objects or updating their type and references,
- Variable affectations: linking variables with values,
- Calls to execute: evaluating expressions as code.

3) Object classifier: The previous classifier has been reused, as in Fig.5. Scripts being based on strings, the address classifier part is unused. The string classifier has been extended to best fit the script particularities, with new constants for the self-reference for example (`"Wscript.ScriptName"`,`"ScriptFullName"`).

5.3 Detection Automata

The transitions corresponding to the different grammar production rules have directly been coded in a prototype similarly to the algorithms from Section 4. Only

two enhancements have been brought to the algorithm in order to increase the performance. A first mechanism avoids duplicate derivations. Coexisting identical derivations artificially increase the number of iterations without identifying other behaviors than the ones already detected. The second enhancement is related to the close and delete interactions. Once again, in order to decrease the number of iterations, the derivations where no interaction intervene between the opening/creation and the closing/deletion of an object, are destroyed. These two mechanisms have proved helpful in regulating the number of parallel derivations.

6 Experimentation and Discussions

For experimentation, hundreds of samples have been gathered, the pool being divided into two categories: Portable Executables and Visual Basic Scripts. For each category, about 50 legitimate samples and 200 malware were considered. According to the repartition in Fig.7, different types of legitimate applications, selected from an healthy system, and malware, downloaded from repositories [19,20], have been considered.

1) Coverage: The experimentation has provided significant detection rates with 51% for PE executables and up to 89% for VB Scripts. Results, behavior by behavior, are described in Tables 2 and 3. Duplication is the most significant malicious behavior. However the additional behaviors, and in particular residency, have detected additional malware where duplication was missed. False positives are almost inexistent according to Tables 4 and 5. The only false positive, related to residency, can be easily explained: the script was a malware cleaner reinitializing the browser start page to clear the infection. On the opposite, important false negative spikes can be localized in the PE results (Table 2): the low detection rates for duplication of Viruses and propagation of Net/Mail Worms are explained by limitations in the collection mechanisms that are assessed in 2).

Comparing VB scripts and PE traces, the false negatives are fewer for VB scripts. Path exploration and affectation monitoring implemented in the analyzer provide a greater coverage. The remaining false negatives are explained by the encryption of the whole malware body which is not supported yet and the cohabitation in a same script of JavaScript and VBScript which makes the syntactic analysis fail. Code localization mechanism could solve the problem. For the analyzer of process traces, the detection rates observed for duplication are consistent with existing works [5]. The real enhancements are twofolds: the

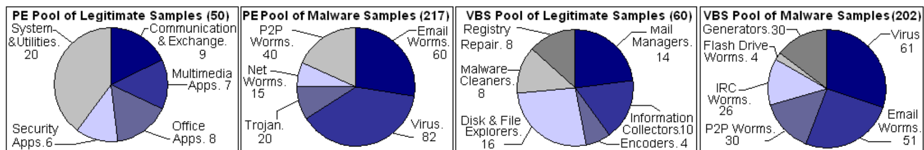


Fig. 7. Repartition of the test pool

Table 2. PE Malware detection. (EmW = Email Worms, P2PW = P2P Worms, V = Virii, NtW = Net Worms, Trj = Trojans, Eng = Functional Polymorphic Engine)

Behaviors	EmW	P2PW	V	NtW	Trj	Global	Eng
Duplication	41(68,33%)	31(77,5%)	15(18,29%)	8(53,33%)	6(30%)	46,54%	30(100%)
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	8(26,67%)
single read/write	41(68,33%)	30(75%)	14(17,07%)	8(53,33%)	6(30%)	45,63%	12(40%)
interleaved r/w	9(15%)	3(7,5%)	3(3,66%)	3(0,2%)	0(0%)	8,29%	10(33,3%)
Propagation	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,44%	17(56,7%)
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
single read/write	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,44%	17(56,7%)
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
Residency	36(60%)	22(55%)	5(60,98%)	6(40%)	9(45%)	35,94%	30(100%)
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
Global detection	43(71,67%)	33(82,50%)	16(19,51%)	8(53,33%)	11(55,00%)	51,15%	30(100%)

Table 3. VBS Malware detection. (EmW = Email Worms, FdW = Flash Drive Worms, IrcW = IRC Worms, P2PW = P2P Worms, V = Viruses, Gen = Generators variants)

Behaviors	EmW	FdW	IrcW	P2PW	V	Gen	Global
Nb string ciphered	1/51	0/4	1/26	0/30	3/61	10/30	15/202
Nb body ciphered	4/51	0/4	0/26	1/30	2/61	0/30	7/202
String encryption	1(100%)	0	0	0(0%)	2(66,67%)	10(100%)	86,67%
Duplication	43(84,31%)	4(100%)	20(76,96%)	22(73,33%)	44(72,13%)	30(100%)	80,70%
direct copy	41(80,39%)	4(100%)	20(76,96%)	22(73,33%)	25(40,98%)	30(100%)	70,30%
single read/write	8(15,69%)	0(0%)	4(15,38%)	3(10%)	21(34,43%)	0(0%)	17,82%
interleaved r/w	1(1,96%)	0(0%)	0(0%)	0(0%)	8(13,11%)	0(0%)	4,46%
Propagation	33(64,71%)	3(75%)	5(19,23%)	25(83,33%)	5(8,20%)	30(100%)	49,99%
direct copy	33(64,71%)	3(75%)	4(15,38%)	25(83,33%)	3(4,92%)	30(100%)	48,52%
single read/write	3(5,88%)	0(0%)	2(7,69%)	1(3,33%)	2(3,28%)	0(0%)	3,96%
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	32(62,75%)	4(100%)	20(76,92%)	18(60,00%)	20(32,79%)	30(100%)	61,39%
Overinfection test	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
conditional	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	46(90,20%)	4(100%)	25(96,15%)	27(90,00%)	50(81,97%)	30(100%)	90,09%

Table 4. PE Legitimate Samples. (Com=Communication & Exchange Applications, MM=Multimedia Apps, Off=Office Apps, Sec=Security Tools, SysU=System & Utilities)

Behaviors	PE ComE	PE MM	PE Off	PE Sec	PE SysU	PE Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%

Table 5. VBS Legitimate Samples. (EmM=Email Managers, InfC=Information Collectors, Enc=Encoders, DfE=Disk & File Explorers, MwC=Malware Cleaners, RegR=Registry Repairs)

Behaviors	VBS EmM	VBS InfC	VBS Enc	VBS DfE	VBS MwC	VBS RegR	VBS Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	1(12,50%)	0(0%)	1,67%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	1(12,5%)	0(0%)	1,67%

parallel detection of additional behaviors described in the same language, and the possibility to feed detection with traces from other sources such as those coming from the script analyzer. Additional tests in Table 2 have been led using the functional polymorphic engine from [9]. For comparison with actual antivirus products, confronted to the engine, the detection rates were between 15% for most of them, up to 90% for others, but with numerous false positives.

2) Limitations in trace collection: A significant part of the missed behaviors are due to limitations in the collection coverage. However, thanks to the layer-based approach, collection and abstraction can be improved for a given platform or language without modifying the upper detection layer.

With regards to dynamic analysis (PE traces), the first reason for missed detections is related to the configuration of the simulated environment. The simulation must seem as real as possible to satisfy the execution conditions of malware. Problems can reside in the software configuration. 65% of the tested Viruses (53/82) did not execute properly: invalid PE, access violations, exceptions. These failures may be explained by the detection of virtualization or anti-debug techniques thwarting dynamic analysis. Problems can also come from the simulated network. Considering worms, their propagation is conditioned by the network configuration. 75% of the Mail Worms (45/60) did not show any SMTP activity because of unreachable servers. Likewise, Net Worms propagate through vulnerabilities only if a vulnerable target is reachable, explaining that 93% of them did not propagate (14/15). All actions conditioned by the simulation configuration are difficult to observe: a potential solution could be forced branching. Notice that this discussion makes sense for off-line analysis but is less of a problem in real-time conditions where we are only interested in the malicious actions effectively performed.

Beyond configuration, the level of the collection can also explain the failures. With a low level collection mechanism, the visibility over the performed actions and the data flow is increased. All flow-sensitive behaviors such as duplication can be missed because of breakdowns in the data flow. Such breakdowns can find their origin sometimes in non monitored system calls and for the most part in the intervention of intermediate buffers where all operations are executed in memory. These buffers are often used in code mutation (polymorphism, metamorphism). 12% of additional virus duplications (10/82) were missed because of data flow breakdowns. The problem is identical with Mail Worms where 8% of the propagations (5/60) were missed because of intermediate buffers intervening in the Base64 encoding. These problems do not come from the behavioral descriptions but from NtTrace which does not capture processor instructions. More complete collection tools either collecting instructions [21] or deploying tainting techniques [22] could avoid these breakdowns in the data flow.

With regards to static analysis (VB scripts), the interpreted language implies a different context where branching exploration is feasible and the whole data flow is observable. Implemented in the script analyzer, these features compensate for the drawbacks of NtTrace and eventually result in better detection rates. However, contrary to the restricted number of system calls, VBScript offers numerous

services. A same operation can be achieved using different managers or interfacing with different Microsoft applications. Additional features could be monitored for a greater coverage: accesses to Messenger, support of the Windows Management Instrumentation (WMI). Moreover, like any other static analysis, script analysis is hindered by encryption and obfuscation. The current version of the analyzer only partially handles these techniques; code encryption is missing for example. Static analysis of scripts is nevertheless easier to consider because no prior disassembly is required and some security locks ease the analysis.

3) Behavior relevance: In addition to data collection, the behavioral model itself must be assessed. The relevance of each behavior must be individually assessed by checking the coverage of its grammatical model. Some behaviors such as duplication, propagation and residency are obviously characteristic to malware. Duplication and propagation are discriminating enough for detection. On the other hand, residency is likely to occur in legitimate programs, during installations for example. To avoid certain false positives, its description could be refined , using additional constraints on the value written to the booting object: the value should refer to the program itself or to one of its duplicated versions. On the other hand, the overinfection model does not seem completely relevant. The problem comes from a description that includes too many restraints limiting its detection. In particular, the conditional structure intervening in the model can not be detected in system call traces. Its generalization could increase detection but the risk of confusion with legitimate error handling would also increase.

4) Performance: Table 6 measures the performances of the different prototype components. Considering abstraction, the analysis of PE traces is the most time consuming. The analyzer uses lots of string comparisons which could be avoided by replacing the off-line analysis by hooking in rel-time for immediate translation. On the other hand, the VBScript analyzer offers satisfying performances. With regards to the detection automata, the performances are also satisfying compared with the worst case complexity. The detection speed remains far below a half second in more than 90% of the cases; the remaining 10% were all malware. The implementation has also revealed that the required space for the derivation stacks was very low, with a maximal stack size of 7. In addition, the number of ambiguities has been measured. If n_e denotes the number of events and n_a the number of ambiguities, in the worst case, we would have $n_a = 2^{n_e}$. But by experience: $n_a << 2^{n_e} \quad and \quad n_a << n_e^2 \quad and \quad n_a \approx \alpha n_e$.

This approximation provides a practical complexity in $\vartheta(k\alpha(\frac{n^2+n}{2}))$ **which is more worth considering.** Moreover, the algorithm can easily be parallelized in multi-core architectures. Figures 8 and 9 provide graphs of the collected α ratios. **It can be observed that above a certain threshold, an important ambiguity ratio** α **is already a sign of malicious activity**.

Table 6. Compared performances on mono and multi-core architectures

NtTrace Analyzer	
Data reduction from PE traces to logs	
Total size: 351Mo Reduced logs: 11Mo	Average: 1,3Mo/Trace Reduction ratio: 29
Execution speed	
Core M 1,4GHz	Dual core 2,6GHz
1,48 s/trace	0,34 s/trace
VB Script Analyzer	
Data reduction from VB scripts to logs	
Total size: 1842Ko Reduced logs: 298Ko	Average: 7Ko/Script Reduction ratio: 6
Execution speed	
Core M 1,4GHz	Dual core 2,6GHz
0,042 s/script +0,50 s/ciphered line	0,016 s/script +0,21 s/ciphered line
Detection Automata	
Execution speed	
Core M 1,4GHz	Dual core 2,6GHz
NT: 0,44 s/log VBS: 0,002 s/log	NT: 0,14 s/log VBS: <0,001 s/log

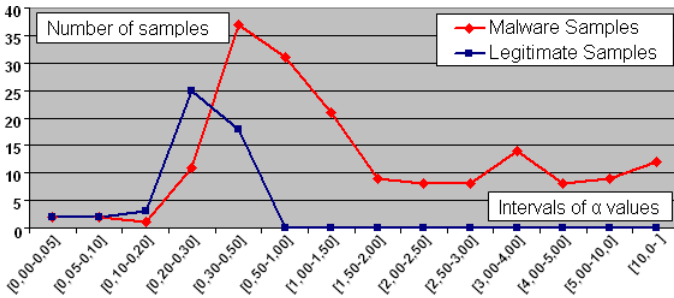


Fig. 8. Ambiguity ratios (α) for PE samples

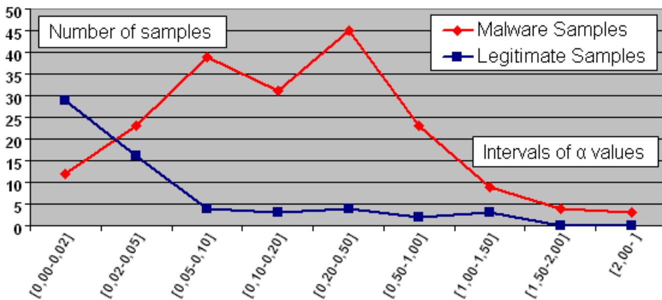


Fig. 9. Ambiguity ratios (α) for VB scripts

7 Conclusions

Detection by attribute automata provides a good coverage of malware using known techniques with 51% of detected PE malware and 89% of VB Scripts malware. The grammatical approach offers a synthetic vision of malicious behaviors. Indeed, only four generic, human-readable, behavioral descriptions have resulted in these detection rates. Unknown malware using variations from these known behaviors should be detected thanks to the abstraction process. In case of innovative techniques, this approach eases the update process. The segmentation between abstraction and detection enables independent updates: in the grammatical descriptions for generic procedures (infrequent), or in the abstraction components for vulnerable objects and APIs. Up until now, the generation of the behavioral descriptions is still manual but the process could be combined with the identification of malicious behaviors by differential analysis proposed by Christodorescu et al. [4]. The experimentations have also stressed the importance of data collection in the detection process. Collection mechanisms are already an active research field and future work can be testing more adapted collection tools deploying tainting.

References

1. Charlier, B.L., Mounji, A., Swimmer, M.: Dynamic detection and classification of computer viruses using general behaviour patterns. Virus Bulletin (1995)
2. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the Network and Distributed System Security Symposium, NDSS (2005)
3. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: Proc. of the IEEE Symposium on Security and Privacy (SSP), pp. 48–62 (2006)
4. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behaviour. In: Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineeering, pp. 5–14 (2007)
5. Morales, J.A., Clarke, P.J., Deng, Y.: Identification of file infecting viruses through detection of self-reference replication. Journal in Computer Virology Online (2008)
6. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A layered architecture for detecting malicious behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
7. Knuth, D.E.: Semantics of context-free grammars. Theory of Computing Systems 2, 127–145 (1968)
8. Jacob, G., Filiol, E., Debar, H.: Malwares as interactive machines: A new framework for behavior modelling. Journal in Computer Virology 4(3), 235–250 (2008)
9. Jacob, G., Filiol, E., Debar, H.: Functional polymorphic engines: Formalisation, implementation and use cases. Journal in Computer Virology Online (2008)
10. US Department of Defense: "Orange Book" - Trusted Computer System Evaluation Criteria. Rainbow Series (1983)
11. NTInternals: The undocumented functions microsoft windows nt/2k/xp/2003, http://undocumented.ntinternals.net

12. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003)
13. Cuppens, F., Miège, A.: Alert correlation in a cooperative intrusion detection framework. In: Proc. of the IEEE Symposium on Security and Privacy (SSP), p. 202 (2002)
14. Al-Mamory, S.O., Zhang, H.: Ids alerts correlation using grammar-based approach. Journal in Computer Virology Online (2008)
15. NtTrace: Native api tracing for windows, http://www.howzatt.demon.co.uk/NtTrace/
16. QEMU: Processor emulator, http://fabrice.bellard.free.fr/qemu/
17. Marion, J.Y., Reynaud-Plantey, D.: Practical obfuscation by interpretation. In: 3rd Workshop on the Theory of Computer Viruses, WTCV (2008)
18. MSDN: Vbscript language reference, http://msdn.microsoft.com/en-us/library/d1wf56tt.aspx
19. VXHeaven: Repository, http://vx.netlux.org/
20. OffensiveComputing: Repository, http://www.offensivecomputing.net/
21. Carrera, E.: Malware - behavior, tools, scripting and advanced analysis. In: HITB-Sec Conf. (2008)
22. Anubis: Analyzing unknown malware, http://anubis.iseclab.org/

Automatic Generation of String Signatures for Malware Detection

Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh

Symantec Research Laboratories

Abstract. Scanning files for signatures is a proven technology, but exponential growth in unique malware programs has caused an explosion in signature database sizes. One solution to this problem is to use *string signatures*, each of which is a contiguous byte sequence that potentially can match many variants of a malware family. However, it is not clear how to automatically generate these string signatures with a sufficiently low false positive rate. Hancock is the first string signature generation system that takes on this challenge on a large scale.

To minimize the false positive rate, Hancock features a scalable model that estimates the occurrence probability of arbitrary byte sequences in goodware programs, a set of library code identification techniques, and diversity-based heuristics that ensure the contexts in which a signature is embedded in containing malware files are similar to one another. With these techniques combined, Hancock is able to automatically generate string signatures with a false positive rate below 0.1%.

Keywords: malware signatures, signature generation, Markov model, library function identification, diversity-based heuristics.

1 Introduction

Symantec's anti-malware response group receives malware samples submitted by its customers and competitors, analyzes them, and creates signatures that could be used to identify instances of them in the field. The number of unique malware samples that Symantec receives has grown exponentially in the recent years, because malware programs are increasingly customized, targeted, and intentionally restricted in distribution scope. The total number of distinct malware samples that Symantec observed in 2008 exceeds 1 million, which is more than the combined sum of all previous years.

Although less proactive than desired, signature-based malware scanning is still the dominant approach to identifying malware samples in the wild because of its extremely low false positive (FP) rate, i.e., the probability of mistaking a goodware program for a malware program is very low. For example, the FP rate requirement for Symantec's anti-malware signatures is below 0.1%. Most signatures used in existing signature-based malware scanners are *hash* signatures, each of which is the hash of a malware file. Although hash signatures have a low false positive rate, the number of malware samples covered by each hash

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 101–120, 2009.
© Springer-Verlag Berlin Heidelberg 2009

signature is also low – typically one. As a result, the total size of the hash signature set grows with the exponential growth in the number of unique malware samples. This creates a signature distribution problem for Symantec: How can we distribute these hash-based malware signatures to hundreds of millions of users across the world several dozen times per day in a scalable way?

One possible solution is to replace hash signatures with *string* signatures, each of which corresponds to a short, contiguous byte sequence from a malware binary. Thus, each string signature can cover many malware files. Traditionally, string signatures are created manually because it is difficult to automatically determine which byte sequence in a malware binary is less FP-prone, i.e., unlikely to appear in any goodware program in the world. Even for manually created string signatures, it is generally straightforward for malware authors to evade them, because they typically correspond to easy-to-modify data strings in malware binaries, such as names of malware authors, special pop-up messages, etc.

Hancock is an automatic string signature generation system developed in Symantec Research Labs that automatically generates high-quality string signatures with minimal FPs and maximal malware coverage. i.e. The probability that a Hancock-generated string signature appears in any goodware program should be very, very low. At the same time each Hancock-generated string signature should identify as many malware programs as possible. Thus, although one string signature takes more space than one hash signature, it uses far less space than all of the hash signatures it replaces.

Given a set of malware samples, Hancock is designed to create a minimal set of N-byte sequences, each of which has a sufficiently low false positive rate, that collectively cover as large a portion of the malware set as possible. Based on previous empirical studies, Hancock sets N to 48. It uses three types of heuristics to test a candidate signature's FP rate: probability-based, disassembly-based, and diversity-based. The first two filter candidate signatures extracted from malware files and the last selects good signatures from among these candidates.

Hancock begins by recursively unpacking malware files using Symantec's unpacking engine. It rejects files that are packed and cannot be unpacked, according to this engine, PEiD [1], and entropy analysis, and stores 48-byte sequences from these files in a list of invalid signatures. Hancock does this because signatures produced on packed files are likely to cover the unpacking code. Blacklisting certain packers should only be done explicitly by a human, rather than through automated signature generation.

Hancock then examines every 48-byte code sequence in unpacked malware files. It finds candidate signatures using probability-based and disassembly-based heuristics: it filters out byte sequences whose estimated occurrence probability in goodware programs, according to a pre-computed goodware model, is above a certain threshold; that are considered a part of library functions; or whose assembly instructions are not sufficiently interesting or unique, based on heuristics that encode malware analysts' selection criteria. It examines only code so that disassembly-based heuristics can work and because malware authors can more easily vary data.

Among those candidate signatures that pass the initial filtering step, Hancock further applies a set of selection rules based on the *diversity* principle: If the set of malware samples containing a candidate signature are similar, then they are less FP-prone. A candidate signature in a diverse set of malware files is more likely to be a part of a library used by several malware families. Though identifying several malware families seems like a good idea, if a signature is part of library code, goodware files might use the same library. On the other hand, if the malware files are similar, they are more likely to belong to one family and the candidate signature is more likely to be code that is unique to that family.

Finally, Hancock is extended to generate string signatures that consist of multiple disjoint byte sequences rather than only one contiguous byte sequence. Although multi-component string signatures are more effective than single-component signatures, they also incur higher run-time performance overhead because individual components are more likely to match goodware programs. In the following sections, we will describe the signature filter algorithms, the signature selection algorithms, and the multi-component generalization used in Hancock.

2 Related Work

Modern anti-virus software typically employ a variety of methods to detect malware programs, such as signature-based scanning [2], heuristic-based detection [3], and behavioral detection [4]. Although less proactive, signature-based malware scanning is still the most prevalent approach to identify malware because of its efficiency and low false positive rate. Traditionally, the malware signatures are created manually, which is both slow and error-prone. As a result, efficient generation of malware signatures has become a major challenge for anti-virus companies to handle the exponential growth of unique malware files. To solve this problem, several automatic signature generation approaches have been proposed.

Most previous work focused on creating signatures that are used by Network Intrusion Detection Systems (NIDS) to detect network worms. Singh et al. proposed EarlyBird [5], which used packet content prevalence and address dispersion to automatically generate worm signatures from the invariant portions of worm payloads. Autograph [6] exploited a similar idea to create worm signatures by dividing each suspicious network flow into blocks terminated by some breakmark and then analyzing the prevalence of each content block. The suspicious flows are selected by a port-scanning flow classifier to reduce false positives. Kreibich and Crowcroft developed Honeycomb [7], a system that uses honeypots to gather inherently suspicious traffic and generates signatured by applying the longest common substring (LCS) algorithm to search for similarities in the packet payloads. One potential drawback of signatures generated from previous approaches is that they are all continuous strings and may fail to match polymorphic worm payloads. Polygraph [8] instead searched for invariant content in the network flows and created signatures consisting of multiple disjoint content substrings.

Polygraph also utilized a naive Bayes classifier to allow the probabilistic matching and classification, and thus provided better proactive detection capabilities. Li et al. proposed Hasma [9], a system that used a model-based algorithm to analyze the invariant contents of polymorphic worms and analytically prove the attack-resilience of generated signatures. PDAS (Position-Aware Distribution Signatures) [10] took advantage of a statistical anomaly-based approach to improve the resilience of signatures to polymorphic malware variants. Another common method for detecting polymorphic malware is to incorporate semantics-awareness into signatures. For example, Christodorescu et al. proposed static semantics-aware malware detection in [11]. They applied a matching algorithm on the disassembled binaries to find the instruction sequences that match the manually generated templates of malicious behaviors, e.g., decryption loop. Yegneswaran et al. developed Nemean [12], a framework for automatic generation of intrusion signatures from honeynet packet traces. Nemean applied clustering techniques on connections and sessions to create protocol-semantic-aware signatures, thereby reducing the possibility of false alarms.

Hancock differs from previous work by focusing on automatically generating high-coverage string signatures with extremely low false positives. Our research was based loosely on the virus signature extraction work [13] by Kephart and Arnold, which was commercially used by IBM. They used a 5-gram Markov chain model of good software to estimate the probability that a given byte sequence would show up in good software. They tested hand-generated signatures and found that it was quite easy to set a model probability threshold with a zero false positive rate and a modest false negative rate (the fraction of rejected signatures that would not be found in goodware) of 48%. They also generated signatures from assembly code (as Hancock does), rather than data, and identified candidate signatures by running the malware in a test environment. Hancock does not do this, as dynamic analysis is very slow in large-scale applications.

Symantec acquired this technology from IBM in the mid-90s and found that it led to many false positives. The Symantec engineers believed that it worked well for IBM because IBM's anti-virus technology was used mainly in corporate environments, making it much easier for IBM to collect a representative set of goodware. By contrast, signatures generated by Hancock are mainly for home users, who have a much broader set of goodware. The model's training set cannot possibly contain, or even represent, all of this goodware. This poses a significant challenge for Hancock in avoiding FP-prone signatures.

3 Signature Candidate Selection

3.1 Goodware Modeling

The first line of defense in Hancock is a Markov chain-based model that is trained on a large goodware set and is designed to estimate the probability of a given byte sequence appearing in goodware. If the probability of a candidate signature appearing in some goodware program is higher than a threshold, Hancock rejects

it. Compared with standard Markov models, Hancock's goodware model has two important features:

- **Scalable to very large goodware set.** Symantec regularly tests its anti-virus signatures against several terabytes of goodware programs. A standard Markov model uses linear space [14] in the training set size, with a large constant factor. Hancock's goodware model focuses only on high-information-density byte sequences so as to scale to very large goodware training sets.
- **Focusing on rare byte sequences.** For a candidate signature to not cause a false positive, its probability of appearing in goodware must be very, very low. Therefore, the primary goal of Hancock's model is to distinguish between low-probability byte sequences and very rare byte sequences.

Basic Algorithm. The model used in Hancock is a fixed-order 5-gram Markov chain model, which estimates the probability of the fifth byte conditioned on the occurrence of the preceding four bytes. Training consists of counting instances of 5-grams – 5-byte sequences – as well as 4-grams, 3-grams, etc. The model calculates the probability of a 48-byte sequence by multiplying estimated probabilities of each of the 48 bytes. A single byte's probability is the probability of that byte following the four preceding bytes. For example, the probability that "e" follows "abcd" is

$$p(\mathrm{e}|\mathrm{abcd}) = \frac{count(\mathrm{abcde})}{count(\mathrm{abcd})} * (1 - \epsilon(count(\mathrm{abcd}))) + p(\mathrm{e}|\mathrm{bcd}) * \epsilon(count(\mathrm{abcd}))$$

In this equation, $count(s)$ is the number of occurrences of the byte sequence s in the training set. We limit overtraining with $\epsilon(count(s))$, the *escape mass* of s. Escape mass decreases with count. Empirically, we found that a good escape mass for our model is $\epsilon(c) = \frac{\sqrt{32}}{\sqrt{32}+\sqrt{c}}$.

Model Pruning. The memory required for a vanilla fixed-order 5-gram model is significantly greater than the size of the original training set. Hancock reduces the memory requirement of the model by incorporating an algorithm that prunes away less useful grams in the model. The algorithm looks at the *relative information gain* of a gram and eliminates it if its information gain is too low. This allows Hancock to keep the most valuable grams, given a fixed memory constraint.

Consider a model's grams viewed as nodes in a tree. The algorithm considers every node X, corresponding to byte sequence s, whose children (corresponding to $s\sigma$ for some byte σ) are all leaves. Let s' be s with its first byte removed. For example, if s is "abcd", s' is "bcd". For each child of X, σ, the algorithm compares $p(\sigma|s)$ to $p(\sigma|s')$. In this example, the algorithm compares $p(\mathrm{e}|\mathrm{abcd})$ to $p(\mathrm{e}|\mathrm{bcd})$, $p(\mathrm{f}|\mathrm{abcd})$ to $p(\mathrm{f}|\mathrm{bcd})$, etc. If the difference between $p(\sigma|s)$ and $p(\sigma|s')$ is smaller than a threshold, that means that X is does not add that much value to σ's probability and the node σ can be pruned away without compromising the model's accuracy.

To focus on low-probability sequences, Hancock uses the difference between the logs of these two probabilities, rather than that between their raw probability values. Given a space budget, Hancock keeps adjusting the threshold until it hits the space target.

Model Merging. Creating a pruned model requires a large amount of intermediate memory, before the pruning step. Thus, the amount of available memory limits the size of the model that can be created. To get around this limit, Hancock creates several smaller models on subsets of the training data, prunes them, and then merges them.

Merging a model M_1 with an existing model M_2 is mostly a matter of adding up their gram counts. The challenge is in dealing with grams pruned from M_1 that exist in M_2 (and vice versa). The merging algorithm must recreate these gram counts in M_1. Let $s\sigma$ be such a gram and let s' be s with its first byte removed. The algorithm estimates the count for $s\sigma$ as $count(s) * p(\sigma|s')$. Once these pruned grams are reconstituted, the algorithm simply adds the two models' gram counts.

Experimental Results. We created an occurrence probability model from a 1-GByte training goodware set and computed the probability of a large number of 24-byte test sequences, extracted from malware files. We checked each test byte sequence against a goodware database, which is a large superset of the training set, to determine if it is a true positive (a good signature) or a false positive (which occurs in goodware). In Figure 1, each point in the FP and TP curves represents the fraction (Y axis value) of test byte sequences whose model probability is below the X axis value.

As expected, TP signatures have much lower probabilities, on average, than FP signatures. A small number of FP signatures have very low probabilities – below 10^{-60}. Around probability 10^{-40}, however, the model does provide excellent discrimination power, rejecting 99% of FP signatures and accepting almost half of TP signatures.

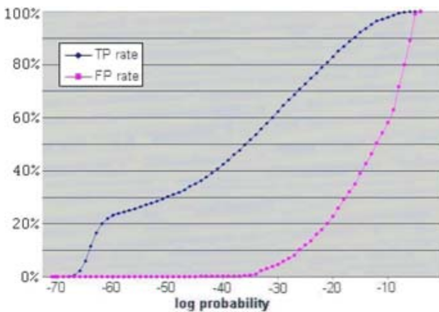


Fig. 1. Fractions of FP and TP sequences with probabilities below the X value

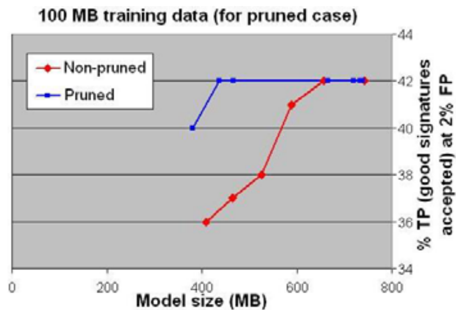


Fig. 2. TP rate comparison between models with varying pruning thresholds and varying training set sizes

To evaluate the effectiveness of Hancock's information gain-based pruning algorithm, we used two sets of models: non-pruned and pruned. The former were trained on 50 to 100 Mbytes of goodware. The latter were trained on 100 Mbytes of goodware and pruned to various sizes. For each model, we then computed its TP rate at the probability threshold that yields a 2% FP rate. Figure 2 shows these TP rates of goodware models versus the model's size in memory. In this case, pruning can roughly halve the goodware model size while offering the same TP rate as the pruned model derived from the same training set.

3.2 Library Function Recognition

A library is a collection of standard functions that implement common operations, such as file IO, string manipulation, and graphics. Modern malware authors use library functions extensively to simplify development, just like goodware authors. By construction, variants of a malware family are likely to share some library functions. Because these library functions also have a high probability of appearing in goodware, Hancock needs to remove them from consideration when generating string signatures. Toward this goal, we developed a set of library function recognition techniques to determine whether a function in a malware file is likely to be a library function or not.

A popular library identification technique is IDA Pro's Fast Library Identification and Recognition Technology (FLIRT) [15], which uses byte pattern matching algorithms (similar to string signature scanning) to quickly determine whether a disassembled function matches any of the signatures known to IDA Pro.[1] Although FLIRT is very accurate in pinpointing common library functions, it still needs some improvement to suit Hancock's needs. First, FLIRT is designed to never falsely identify a library. To achieve this, FLIRT first tries to identify the compiler type (e.g., Visual C++ 7.0, 8.0, Borland C++, Delphi, etc.) of a disassembled program and applies only signatures for that compiler. For example, vcseh signatures (Structured Exception Handling library signatures) will only be applied to binary files that appear to have been compiled with Visual C++ 7 or 8. This conservative approach can lead to false negatives (a library function not identified) because of failure in correctly detecting the compiler type. In addition, because FLIRT uses a rigorous pattern matching algorithm to search for signatures, small variation in libraries, e.g., minor changes in the source code, different settings in compiler optimization options or use of different compiler versions to build the library, could prevent FLIRT from recognizing all library functions in a disassembled program.

In contrast to FLIRT's conservative approach, Hancock's primary goal is to eliminate false positive signatures. It takes a more aggressive stance by being willing to mistake non-library functions for library functions. Such misidentification is acceptable because it prevents any byte sequence that is potentially

[1] IDA Pro ships with a database of signatures for about 120 libraries associated with common compilers. Each signature corresponds to a binary pattern in a library function.

associated with a library function from being used as a malware signature. We exploited this additional latitude with the following three heuristics:

Universal FLIRT Heuristic. This heuristic generalizes IDA Pro's FLIRT technique by matching a given function against all FLIRT signatures, regardless of whether they are associated with the compiler used to compile the function. This generalization is useful because malware authors often post-process their malware programs to hide or obfuscate compiler information in an attempt to deter any reverse engineering efforts. Moreover, any string signatures extracted from a function in a program compiled by a compiler C1 that looks like a library function in another compiler C2 are likely to cause false positives against programs compiled by C2 and thus should be rejected.

Library Function Reference Heuristic. This heuristic identifies a library function if the function is statically called, directly or indirectly, by any known library function. The rationale behind this heuristic is that since a library cannot know in advance which user program it will be linked to, it is impossible for a library function to statically call any user-written function, except callback functions, which are implemented through function pointers and dynamically resolved. As a result, it is safe to mark all children of a library function in its call tree as library functions. Specifically, the proposed technique disassembles a binary program, builds a function call graph representation of the program, and marks any function that is called by a known library function as a library function. This marking process repeats itself until no new library function can be found.

In general, compilers automatically include into an executable binary certain template code, such as startup functions or error handling, which IDA Pro also considers as library functions as well. These template functions and their callees must be excluded in the above library function marking algorithm. For example, the entry point function *start* and *mainCRTstartup* in Visual C++-compiled binaries are created by the compiler to perform startup preparation (e.g., execute global constructors, catch all uncaught exceptions) before invoking the user-defined main function.

3.3 Code Interestingness Check

The *code interestingness* check is designed to capture the intuitions of Symantec's malware analysis experts about what makes a good string signature. For the most part, these metrics identify signatures that are less likely to be false positives. They can also identify malicious behavior, though avoiding false positives is the main goal. The code interestingness check assigns a score for each "interesting" instruction pattern appearing in a candidate signature, sums up these scores, and rejects the candidate signature if its sum is below a threshold, i.e. not interesting enough. The interesting patterns used in Hancock are:

- **Unusual constant values.** Constants sometimes have hard-coded values that are important to malware, such as the IP address and port of a command

and control server. More importantly, if a signature has unusual constant values, it is less likely to be a false positive.

- **Unusual address offsets.** Access to memory that is more than 32 bytes from the base pointer can indicate access to a large class or structure. If these structures are unique to a malware family, then accesses to particular offsets into this structure are less likely to show up in goodware. This pattern is not uncommon among legitimate Win32 applications. Nonetheless, it has good discrimination power.
- **Local or non-library function calls.** A local function call itself is not very distinctive, but the setup for local function calls often is, in terms of how it is used and how its parameters are prepared. In contrast, setup for system calls is not as interesting, because they are used in many programs and invoked in a similar way.
- **Math instructions.** A malware analyst at Symantec noted that malware often perform strange mathematical operations, to obfuscate and for various other reasons. Thus, Hancock looks for strange sequences of XORs, ADDs, etc. that are unlikely to show up in goodware.

4 Signature Candidate Filtering

Hancock selects candidate signatures using techniques that assess a candidate's FP probability based solely on its contents. In this section, we describe a set of filtering techniques that remove from further consideration those candidate signatures that are likely to cause a false positive based on the signatures' use in malware files.

These *diversity-based* techniques only accept a signature if it matches variants of one malware family (or a small number of families). This is because, if a byte sequence exists in many malware families, it is more likely to be library code – code that goodware could also use. Therefore, malware files covered by a Hancock signature should be similar to one another.

Hancock measures the diversity of a set of binary files based on their byte-level and instruction-level representations. The following two subsections describe these two diversity measurement methods.

4.1 Byte-Level Diversity

Given a signature, S, and the set of files it covers, X, Hancock measures the byte-level similarity or diversity among the files in X by extracting the byte-level context surrounding S and computing the similarity among these contexts. More concretely, Hancock employs the following four types of byte-level signature-containing contexts for diversity measurement.

Malware Group Ratio/Count. Hancock clusters malware files into groups based on their byte-level histogram representation. It then counts the number of groups to which the files in X belong. If this number divided by the number

of files in X exceeds a threshold ratio, or if the number exceeds a threshold count, Hancock rejects S. These files cannot be variants of a single malware family, if each malware group indeed corresponds to a malware family.

Signature Position Deviation. Hancock calculates the position of S within each file in X, and computes the standard deviation of S's positions in these files. If the standard deviation exceeds a threshold, Hancock rejects S, because a large positional deviation suggests that S is included in the files it covers for very different reasons. Therefore, these files are unlikely to belong to the same malware family. The position of S in a malware file can be an absolute byte offset, which is with respect to the beginning of the file, or a relative byte offset, which is with respect to the beginning of the code section containing S.

Multiple Common Signatures. Hancock attempts to find another common signature that is present in all the files in X and is at least 1 Kbyte away from S. If such a common signature indeed exists and the distance between this signature and S has low standard deviation among the files in X, then Hancock accepts S because this suggests the files in X share a large chunk of code and thus are likely to be variants of a single malware family. Intuitively, this heuristic measures the similarity among files in X using additional signatures that are sufficiently far away, and can be generalized to using the third or fourth signature.

Surrounding Context Count. Hancock expands S in each malware file in X by adding bytes to its beginning and end until the resulting byte sequences become different. For each such distinct byte sequence, Hancock repeats the same expansion procedure until the expanded byte sequences reach a size limit, or when the total number of distinct expanded byte sequences exceeds a threshold. If this expansion procedure terminates because the number of distinct expanded byte sequences exceeds a threshold, Hancock rejects S, because the fact that there are more than several distinct contexts surrounding S among the files in X suggests that these files do not belong to the same malware family.

4.2 Instruction-Level Diversity

Although byte-level diversity measurement techniques are easy to compute and quite effective in some cases, they treat bytes in a binary file as numerical values and do not consider their semantics. Given a signature S and the set of files it covers, X, instruction-level diversity measurement techniques, on the other hand, measure the instruction-level similarity or diversity among the files in X by extracting the instruction-level context surrounding S and computing the similarity among these contexts.

Enclosing Function Count. Hancock extracts the enclosing function of S in each malware file in X, and counts the number of distinct enclosing functions. If the number of distinct enclosing functions of S with respect to X is higher than a threshold, Hancock rejects S, because S appears in too many distinct

contexts among the files in X and therefore is not likely to be an intrinsic part of one or a very small number of malware families. To determine if two enclosing functions are distinct, Hancock uses the following three identicalness measures, in decreasing order of strictness:

- The byte sequences of the two enclosing functions are identical.
- The instruction op-code sequences of the two enclosing functions are identical. Hancock extracts the op-code part of every instruction in a function, and normalizes variants of the same op-code class into their canonical op-code. For example, there are about 10 different X86 op-codes for ADD, and Hancock translates all of them into the same op-code. Because each instruction's operands are ignored, this measure is resistant to intentional or accidental polymorphic transformations such as re-locationing, register assignment, etc.
- The instruction op-code sequences of the two enclosing functions are identical after *instruction sequence normalization*. Before comparing two op-code sequences, Hancock performs a set of de-obfuscating normalizations that are designed to undo simple obfuscating transformations, such as replacing "`test esi, esi`" with "`or esi, esi`", replacing "`push ebp; mov ebp, esp`" with "`push ebp; push esp; pop ebp`", etc.

5 Multi-Component String Signature Generation

Traditionally, string signatures used in AV scanners consist of a contiguous sequence of bytes. We refer to these as single-component signature (SCS). A natural generalization of SCS is multi-component signatures (MCS), which consist of multiple byte sequences that are potentially disjoint from one another. For example, we can use a 48-byte SCS to identify a malware program; for the same amount of storage space, we can create a two-component MCS with two 24-byte sequences. Obviously, an N-byte SCS is a special case of a K-component MCS where each component is of size $\frac{N}{K}$. Therefore, given a fixed storage space budget, MCS provides more flexibility in choosing malware-identifying signatures than SCS, and is thus expected to be more effective in improving coverage without increasing the false positive rate.

In the most general form, the components of a MCS do not need to be of the same size. However, to limit the search space, in the Hancock project we explore only those MCSs that have equal-sized components. So the next question is how many components a MCS should have, given a fixed space budget. Intuitively, each component should be sufficiently long so that it is unlikely to match a random byte sequence in binary programs by accident. On the other hand, the larger the number of components in a MCS, the more effective it is in eliminating false positives. Given the above considerations and the practical signature size constraint, Hancock chooses the number of components in each MCS to be between 3 and 5.

Hancock generates the candidate component set using a goodware model and a goodware set. Unlike SCS, candidate components are drawn from both data and code, because intuitively, combinations of code component signatures and

data component signatures make perfectly good MCS signatures. When Hancock examines an $\frac{N}{K}$-byte sequence, it finds the longest substring containing this sequence that is common to all malware files that have the sequence. Hancock takes only one candidate component from this substring. It eliminates all sequences that occur in the goodware set and then takes the sequence with the lowest model probability. Unlike SCS, there is no model probability threshold.

Given a set of qualified component signature candidates, S1, and the set of malware files that each component signature candidate covers, Hancock uses the following algorithm to arrive at the final subset of component signature candidates used to form MCSs, S2:

1. Compute for each component signature candidate in S1 its *effective coverage value*, which is a sum of weights associated with each file the component signature candidate covers. The weight of a covered file is equal to its *coverage count*, the number of candidates in S2 already covering it, except when the number of component signatures in S2 covering that file is larger than or equal to K, in which case the weight is set to zero.
2. Move the component signature candidate with the highest effective coverage value from S1 to S2, and increment the coverage count of each file the component signature candidate covers.
3. If there are still malware files that are still uncovered or there exists at least one component signature in S1 whose effective coverage value is non-zero, go to Step 1; otherwise exit.

The above algorithm is a modified version of the standard greedy algorithm for the *set covering* problem. The only difference is that it gauges the value of each component signature candidate using its effective coverage value, which takes into account the fact that at least K component signatures in S2 must match a malware file before the file is considered covered. The way weights are assigned to partially covered files is meant to reflect the intuition that the value of a component signature candidate to a malware file is higher when it brings the file's coverage count from $X - 1$ to X than that from $X - 2$ to $X - 1$, where X is less than or equal to K.

After S2 is determined, Hancock finalizes the K-component MCS for each malware file considered covered, i.e., whose coverage count is no smaller than K. To do so, Hancock first checks each component signature in S2 against a goodware database, and marks it as an FP if it matches some goodware file in the database. Then Hancock considers all possible K-component MCSs for each malware file and chooses the one with the smallest number of components that are an FP. If the number of FP components in the chosen MCS is higher than a threshold, T_{FP}, the MCS is deemed as unusable and the malware file is considered not covered. Empirically, T is chosen to be 1 or 2. After each malware file's MCS is determined, Hancock applies the same diversity principle to each MCS based on the malware files it covers.

6 Evaluation

6.1 Methodology

To evaluate the overall effectiveness of Hancock, we used it to generate 48-byte string signatures for two sets of malware files, and use the coverage and number of false positives of these signatures as the performance metrics. The first malware set has 2,363 unpacked files that Symantec gathered in August 2008. The other has 46,288 unpacked files (or 112,156 files before unpacking) gathered in 2007-2008. The goodware model used in initial signature candidate filtering is derived from a 31-Gbyte goodware training set. In addition, we used another 1.8-Gbyte goodware set to filter out FP-prone signature candidates. To determine which signatures are FPs, we tested each generated signature against a 213-Gbyte goodware set. The machine used to perform these experiments has four quad-core 1.98-GHz AMD Opteron processors and 128 Gbytes of RAM.

6.2 Single-Component Signatures

Because almost every signature candidate selection and filtering technique in Hancock comes with an empirical threshold parameter, it is impossible to present results corresponding to all possible combinations of these parameters. Instead, we present results corresponding to three representative settings, which are shown in Table 1 and called *Loose*, *Normal* and *Strict*. The generated signatures cover overlapping sets of malware files.

To gain additional assurance that Hancock's FP rate was low enough, Symantec's malware analysts wanted to see not only zero false positives, but also that the signatures look good – they look like they encode non-generic behavior that is unlikely to show up in goodware. To that end, we manually ranked signatures on the August 2008 malware set as good, poor, and bad.

To get a rough indication of the maximum possible coverage, the last lines in tables 2 and 3 show the coverage of all non-FP candidate signatures. The probability-based and dissassembly-based heuristics were still enabled with Loose threshold settings.

These results show not only that Hancock has a low FP rate, but also that tighter thresholds can produce signatures that look less generic. Unfortunately, it can only produce signatures to cover a small fraction of the specified malware.

Several factors limit Hancock's coverage:

– Hancock's packer detection might be insufficient. PEiD recognizes many packers, but by no means all of them. Entropy detection can also be fooled:

Table 1. Heuristic threshold settings

Threshold setting	Model probability	Group ratio	Position deviation	# common signatures	Interestingness	Minimum coverage
Loose	-90	0.35	4000	1	13	3
Normal	-90	0.35	3000	1	14	4
Strict	-90	0.35	3000	2	17	4

Table 2. Results for August 2008 data

Threshold setting	Cover-age	# FPs	Good sig.s	Poor sig.s	Bad sig.s
Loose	15.7%	0	6	7	1
Normal	14.0%	0	6	2	0
Strict	11.7%	0	6	0	0
All non-FP	22.6%	0	10	11	9

Table 3. Results for 2007-8 data

Threshold	Coverage	Sig.s	FPs
Loose	14.1%	1650	7
Normal	11.7%	767	2
Normal, pos. deviation 1000	11.3%	715	0
Strict	4.4%	206	0
All non-FP	31.7%	7305	0

some packers do not compress the original file's data, but only obfuscate it. Diversity-based heuristics will probably reject most candidate signatures extracted from packed files. (Automatically generating signatures for packed files would be bad, anyway, since they would be signatures on packer code.)

– Hancock works best when the malware set has many malware families and many files in each malware family. It needs many families so that diversity-based heuristics can identify generic or rare library code that shows up in several malware families. It needs many files in each family so that diversity-based heuristics can identify which candidate signatures really are characteristic of a malware family. If the malware sets have many malware families with only a few files each, this would lower Hancock's coverage.
– Malware polymorphism hampers Hancock's effectiveness. If only some code is polymorphic, Hancock can still identify high coverage signatures in the remaining code. If the polymorphic code has a relatively small number of variations, Hancock can still identify several signatures with moderate coverage that cover most files in the malware family. If all code is polymorphic, with a high degree of variation, Hancock will cover very few of the files.
– Finally, the extremely stringent FP requirement means setting heuristics to very conservative thresholds. Although the heuristics have good discrimination power, they still eliminate many good signatures. e.g. The group count heuristic clusters malware into families based on a single-byte histogram. This splits most malware families into several groups, with large malware families producing a large number of groups. An ideal signature for this family will occur in all of those groups. Thus, for the sake of overall discrimination power, the group count heuristic will reject all such ideal signatures.

Sensitivity Study. A heuristic's *discrimination power* is a measure of its effectiveness. A heuristic has good discrimination power if the fraction of false positive signatures that it eliminates is higher than the fraction of true positive signatures it eliminates. These results depend strongly on which other heuristics are in use. We tested heuristics in two scenarios: we measured their *raw discrimination power* with other heuristics disabled; and we measured their *marginal discrimination power* with other heuristics enabled with conservative thresholds.

First, using the August 2008 malware set, we tested the raw discrimination power of each heuristic. Table 4 shows the baseline setting, more conservative

Table 4. Raw Discrimination Power

Heuristic	FPs	Cov.	DP
Max pos. deviation (from ∞ to 8,000)	41.7%	96.6%	25
Min file coverage (from 3 to 4)	6.0%	83.3%	15
Group ratio (from 1.0 to .6)	2.4%	74.0%	12
Model log probability (from -80 to -100)	51.2%	73.7%	2.2
Code interestingness (from 13 to 15)	58.3%	78.2%	2.2
Multiple common sig.s (from 1 to 2)	91.7%	70.2%	0.2
Universal FLIRT	33.1%	71.7%	3.3
Library function reference	46.4%	75.7%	2.8
Address space	30.4%	70.8%	3.5

Table 5. Marginal Discrimination Power

Heuristic	FPs	Coverage
Max pos. deviation (from 3,000 to ∞)	10	121%
Min file coverage (from 4 to 3)	2	126%
Group ratio (from 0.35 to 1)	16	162%
Model log probability (from -90 to -80)	1	123%
Code interestingness (from 17 to 13)	2	226%
Multiple common sig.s (from 2 to 1)	0	189%
Universal FLIRT	3	106%
Library function reference	4	108%
Address space	3	109%

setting, and discrimination power for each heuristic. The library heuristics (Universal FLIRT, library function reference, and address space) are enabled for the baseline test and disabled to test their own discrimination powers. Using all baseline settings, the run covered 551 malware files with 220 signatures and 84 false positives. Discrimination power is calculated as $\log \frac{\text{FPs}_i}{\text{FPs}_f} \Big/ \log \frac{\text{Coverage}_i}{\text{Coverage}_f}$.

Table 4 shows most of these heuristics to be quite effective. Position deviation and group ratio have excellent discrimination power (DP); the former lowers coverage very little and the latter eliminates almost all false positives. Model probability and code interestingness showed lower DP because their baseline settings were already somewhat conservative. Had we disabled these heuristics entirely, the baseline results would have been so overwhelmed with false positives as to be meaningless. All four of these heuristics are very effective.

Increasing the minimum number of malware files a signature must cover eliminates many marginal signatures. The main reason is that, for lower coverage numbers, there are so many more candidate signatures that some bad ones will get through. Raising the minimum coverage can have a bigger impact in combination with diversity-based heuristics, because those heuristics work better with more files to analyze.

Requiring two common signatures eliminated more good signatures than false positive signatures. It actually made the signatures, on average, worse.

Finally, the library heuristics all work fairly well. They each eliminate 50% to 70% of false positives while reducing coverage less than 30%. In the test for each library heuristic, the other two library heuristics and basic FLIRT functionality were still enabled. This shows that none of these library heuristics are redundant and that these heuristics go significantly beyond what FLIRT can do.

Marginal Contribution of Each Technique. Then we tested the effectiveness of each heuristic when other heuristics were set to the Strict thresholds from table 1. We tested the tunable heuristics with the 2007-8 malware set with Strict baseline threshold settings from table 1. Testing library heuristics was more computationally intensive (requiring that we reprocess the malware set), so we tested them on August 2008 data with baseline Loose threshold settings. Since both sets of baseline settings yield zero FPs, we decreased each heuristic's threshold (or disabled it) to see how many FPs its conservative setting eliminated and how much it reduced malware coverage. Table 5 shows the baseline and more liberal settings for each heuristic. Using all baseline settings, the run covered 1194 malware files with 206 signatures and 0 false positives.

Table 5 shows that almost all of these heuristics are necessary to reduce the FP rate to zero. Among the tunable heuristics, position deviation performs the best, eliminating the second most FPs with the lowest impact on coverage. The group ratio also performs well. Requiring a second common signature does not seem to help at all. The library heuristics perform very well, barely impacting coverage at all. Other heuristics show significantly decreased marginal discrimination power, which captures an important point: if two heuristics eliminate the same FPs, they will show good raw discrimination power, but poor marginal discrimination power.

6.3 Single-Component Signature Generation Time

The most time-consuming step in Hancock's string signature generation process is goodware model generation, which, for the model used in the above experiments, took approximately one week and used up all 128 GBytes of available memory in the process of its creation. Fortunately, this step only needs to be done once. Because the resulting model is much smaller than the available memory in the testbed machine, using the model to estimate a signature candidate's occurrence probability does not require any disk I/O.

The three high-level steps in Hancock at run time are malware pre-processing (including unpacking and disassembly), picking candidate signatures, and applying diversity-based heuristics to arrive at the best ones. Among them, malware pre-processing is the most expensive step, but is also quite amenable to parallelization. The two main operations in malware pre-processing are recursively unpacking malware files and disassembling both packed and unpacked files using IDA Pro. Both use little memory, so we parallelized them to use 15 of our machines 16 cores. For the 2007-2008 data set, because of the huge number of packed malware files and the decreasing marginal return of analyzing them, Hancock disassembled only 5,506 packed files. Pre-processing took 71 hours.

Picking candidate signatures took 145 minutes and 37.4 GB of RAM. 15 minutes and 34.3 GB of RAM went to loading the goodware model. The remainder was for scanning malware files and picking and storing candidate signatures in memory and then on disk.

Table 6. Multi-Component Signature results

# components	Permitted component FPs	Coverage	# Signatures	# FPs
2	1	28.9%	76	7
2	0	23.3%	52	2
3	1	26.9%	62	1
3	0	24.2%	44	0
4	1	26.2%	54	0
4	0	18.1%	43	0
5	1	26.2%	54	0
5	0	17.9%	43	0
6	1	25.9%	51	0
6	0	17.6%	41	0

Generating the final signature set took 420 minutes and 6.07 GB of RAM. Most of this time was spent running IDA Pro against byte sequences surrounding the final signatures to output their assembly representation. Without this step, the final signature generation step would have taken only a few minutes.

6.4 Multi-Component Signatures

We tested MCS signatures with 2 to 6 components, with each part being 16 bytes long. We used a 3.0 GB goodware set to select component candidates and tested for false positives with a 34.9 GB set of separate goodware.[2] Table 6 shows the coverage and false positive rates when 0 or 1 components could be found in the smaller goodware set.

We first observe that permitting a single component of an MCS to be an FP in our small goodware set consistently results in higher coverage. However, from 2- and 3-component signatures, we also see that allowing a single component FP results in more entire MCS FPs, where all signature components occur in a single goodware file.

We can trade off coverage and FP rate by varying the number of signatures components and permitted component FPs. Three to five part signatures with 0 or 1 allowed FPs seems to provide the best tradeoff between coverage and FPs.

Since we applied so few heuristics to get these results, beyond requiring the existence of the multiple, disjoint signature components which make up the signature, it is perhaps surprising that we have so few MCS FPs. We explain this by observing that although we do not limit MCS components to code bytes, we do apply all the library code reducing heuristics through IDA disassembly described in Section 3.2.

Also, the way in which signature components are selected from contiguous runs of identical bytes may reduce the likelihood of FPs. If a long, identical byte sequence exists in a set of files, the 16 byte signature component with lowest

[2] This final goodware set was smaller than in SCS tests because of the difficulty of identifying shorter, 16-byte sequences.

probability will be selected. Moreover, no other signature component will be selected from the same run of identical bytes. Thus, if malware shares an identical uncommon library (which we fail to identify as a library) linked in contiguously in the executable, at most one signature component will be extracted from this sequence of identical bytes. The other components must come from some other shared code or data.

Finding candidate signatures took 1,278 minutes and 117 GB of RAM. Picking the final signature sets took 5 to 17 minutes and used 9.0 GB of RAM.

7 Discussion

The main limitation of the current version of Hancock is its low coverage, which is also the biggest surprise in this project. One potential explanation for this result is that malware authors have recently evolved their malware distribution strategy from a "few malware families each with many variants" model to a "many malware families each with few variants" model, so as to keep each distributed malware sample effective for as long as possible. Because Hancock is designed to generate string signatures that correspond to common byte sequences shared by variants of the same malware family, if the average number of variants in each family is decreased, it is more difficult for Hancock to generate signature with good coverage while keeping the false positive rate in check, especially when state-of-the-art malware classification technology is still quite primitive.

To generate new malware families, malware authors use sophisticated packing and/or metamorphic transformation tools. The current version of Hancock cannot do much for binaries created by these tools. The static unpack engine Hancock uses is used in Symantec's anti-virus products. Still it cannot handle many packers or metamorphic transformation tools. For example, in the largest test described in Section 6.2, Hancock has to ignore 59% of the input malware set because it found them to be packed and could not unpack them. Among the remaining 41%, some of them are probably packed (perhaps partially), but are not detected by Hancock. For such malware files, Hancock won't create string signatures for them because they do not share common byte sequences with other malware files.

In the future, we plan to incorporate dynamic unpacking techniques, such as Justin [16], to reduce the impact of packers on Hancock's coverage. It is also possible to mitigate the packer problem by blacklisting binaries packed by certain packers. We did not spend much effort investigating metamorphic transformation tools in the Hancock project, because string signature-based malware identification may not be effective for metamorphic binaries. Instead, behavior-based malware identification may be a more promising solution. Nonetheless, systematically studying modern metamorphic tools and devising a taxonomical framework to describe them will be very useful contributions to the field of malware analysis.

Another significant limitation of Hancock is its lack of dynamic analysis, which forces it to give up on packed or metamorphically transformed binaries that it

cannot recognize or restore. The rationale for the design decision of employing only static analysis in Hancock is that it cannot afford the run-time performance cost associated with dynamic analysis given the current and future malware arrival rate. In addition, even state-of-the-art dynamic analysis techniques cannot solve all the packer or metamorphism problems for Hancock.

Although many of Hancock's heuristics can be evaded, in general this is a much smaller concern than the problem that malware authors avoid using known string signatures in their binaries. Attackers can (and do) test newly generated malware files against popular anti-virus products. In contrast, even if malware authors create malware files that do not contain byte sequences that Hancock may use as signatures, there is no easy way to test the effectiveness of these malware files against Hancock's signature generation algorithms, because it is not publicly available and because it has so many empirical built-in parameters. In theory, security by obscurity is not a foolproof solution; in practice, it is very difficult, if not infeasible, to evade Hancock's signature generation heuristics.

8 Conclusion

Given a set of malware files, an ideal string signature generation system should be able to automatically generate signatures in such a way that the number of signatures required to cover the malware set is minimal and the probability of these signatures appearing in goodware programs is also minimal. The main technical challenge of building such string signature generation systems is how to determine how FP-prone a byte sequence is without having access to even a sizeable portion of the world's goodware set. This false positive problem is particularly challenging because the goodware set is constantly growing, and is potentially unbounded. In the Hancock project, we have developed a series of signature selection and filtering techniques that collectively could remove most, if not all, FP-prone signature candidates, while maintaining a reasonable coverage of the input malware set. In summary, the Hancock project has made the following research contributions in the area of malware signature generation:

– A scalable goodware modeling technique that prunes away unimportant nodes according to their relative information gain and merges sub-models so as to scale to very large training goodware sets,
– A set of diversity-based techniques that eliminate signature candidates when the set of malware programs they cover exhibit high diversity, and
– The first known string signature generation system that is capable of creating multi-component string signatures which have been shown to be more effective than single-component string signatures.

Although Hancock represents the state of the art in string signature generation technology, there is still room for further improvement. The overall coverage of Hancock is lower than what we expected when we started the project. How to improve Hancock's coverage without increasing the FP rate of its signatures is worth further research. Although the multi-component signatures that Hancock generates are more effective than single-component signatures, their actual

run-time performance impact is unclear and requires more thorough investigation. Moreover, there could be other forms of multi-component signatures that Hancock does not explore and therefore deserve additional research efforts.

This paper omitted discussion of several additional heuristics explored in project Hancock. See [17] for more details.

References

1. PEiD, http://www.peid.info
2. Clam AntiVirus: Creating signatures for ClamAV (2007), http://www.clamav.net/doc/latest/signatures.pdf
3. Arnold, W., Tesauro, G.: Automatically generated win32 heuristic virus detection. In: Proceedings of Virus Bulletin Conference (2000)
4. Jacob, G., Debar, H., Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. Journal in Computer Virology 4(3) (2008)
5. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: OSDI 2004: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, CA, USA, p. 4. USENIX Association (2004)
6. Kim, H.: Autograph: Toward automated, distributed worm signature detection. In: Proceedings of the 13th Usenix Security Symposium, pp. 271–286 (2004)
7. Kreibich, C., Crowcroft, J.: Honeycomb: creating intrusion detection signatures using honeypots. SIGCOMM Comput. Commun. Rev. 34(1), 51–56 (2004)
8. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: SP 2005: Proceedings of the 2005 IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 226–241. IEEE Computer Society, Los Alamitos (2005)
9. Li, Z., Sanghi, M., Chen, Y., Kao, M., Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In: SP 2006: Proceedings of the 2006 IEEE Symposium on Security and Privacy, Oakland06, pp. 32–47. IEEE Computer Society, Los Alamitos (2006)
10. Tang, Y., Chen, S.: Defending against internet worms: A signature-based approach. In: Proceedings of IEEE INFOCOM 2005 (2005)
11. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware malware detection. In: Proceedings of the IEEE Symposium on Security and Privacy (2005)
12. Yegneswaran, V., Giffin, J.T., Barford, P., Jha, S.: An architecture for generating semantics-aware signatures. In: SSYM 2005: Proceedings of the 14th conference on USENIX Security Symposium, Berkeley, CA, USA, p. 7. USENIX Association (2005)
13. Kephart, J.O., Arnold, W.C.: Automatic extraction of computer virus signatures. In: Proceedings of the 4th Virus Bulletin International Conference (1994)
14. Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order markov models. Journal of Artificial Intelligence Research 22, 384–421 (2004)
15. Guilfanov, I.: Fast library identification and recognition technology (1997), http://www.hex-rays.com/idapro/flirt.htm
16. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 98–115. Springer, Heidelberg (2008)
17. Griffin, K., Schneider, S., Hu, X., Chiueh, T.: Automatic generation of string signatures for malware detection (2009), http://www.ecsl.cs.sunysb.edu/tr/TR236.pdf

PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime

M. Zubair Shafiq[1], S. Momina Tabish[1,2], Fauzan Mirza[2,1], and Muddassar Farooq[1]

[1] Next Generation Intelligent Networks Research Center (nexGIN RC)
National University of Computer & Emerging Sciences (FAST-NUCES)
Islamabad, 44000, Pakistan
{zubair.shafiq,momina.tabish,muddassar.farooq}@nexginrc.org
[2] School of Electrical Engineering & Computer Science (SEECS)
National University of Sciences & Technology (NUST)
Islamabad, 44000, Pakistan
fauzan.mirza@seecs.edu.pk

Abstract. In this paper, we present an accurate and realtime PE-Miner framework that automatically extracts distinguishing features from portable executables (PE) to detect zero-day (i.e. previously unknown) malware. The distinguishing features are extracted using the structural information standardized by the Microsoft Windows operating system for executables, DLLs and object files. We follow a threefold research methodology: (1) identify a set of structural features for PE files which is computable in realtime, (2) use an efficient preprocessor for removing redundancy in the features' set, and (3) select an efficient data mining algorithm for final classification between benign and malicious executables.

We have evaluated PE-Miner on two malware collections, VX Heavens and Malfease datasets which contain about 11 and 5 thousand malicious PE files respectively. The results of our experiments show that PE-Miner achieves more than 99% detection rate with less than 0.5% false alarm rate for distinguishing between benign and malicious executables. PE-Miner has low processing overheads and takes only 0.244 seconds on the average to scan a given PE file. Finally, we evaluate the robustness and reliability of PE-Miner under several regression tests. Our results show that the extracted features are robust to different packing techniques and PE-Miner is also resilient to majority of crafty evasion strategies.

Keywords: Data Mining, Malicious Executable Detection, Malware Detection, Portable Executables, Structural Information.

1 Introduction

A number of non-signature based malware detection techniques have been proposed recently. These techniques mostly use heuristic analysis, behavior analysis, or a combination of both to detect malware. Such techniques are being actively investigated because of their ability to detect zero-day malware without any a

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 121–141, 2009.
© Springer-Verlag Berlin Heidelberg 2009

priori knowledge about them. Some of them have been integrated into the existing Commercial Off the Shelf Anti Virus (COTS AV) products, but have achieved only limited success [26], [13]. The most important shortcoming of these techniques is that they are not *realtime deployable*[1]. We, therefore, believe that the domain of *realtime deployable* non-signature based malware detection techniques is still open to novel research.

Non-signature based malware detection techniques are primarily criticized because of two inherent problems: (1) high *fp* rate, and (2) large processing overheads. Consequently, COTS AV products mostly utilize signature based detection schemes that provide low *fp* rate and have acceptable processing overheads. But it is a well-known fact that signature based malware detection schemes are unable to detect *zero-day* malware. We cite two reports to highlight the alarming rate at which new malware is proliferating. The first report is by Symantec that shows an increase of 468% in the number of malware from 2006 to 2007 [25]. The second report shows that the number of malware produced in 2007 alone was more than the total number of malware produced in the last 20 years [6]. These surveys suggest that signature based techniques cannot keep abreast with the security challenges of the new millennium because not only the size of the signatures' database will exponentially increase but also the time of matching signatures. These bottlenecks are even more relevant on resource constrained smart phones and mobile devices [3]. We, therefore, envision that in near future signature based malware detection schemes will not be able to meet the criterion of *realtime deployable* as well.

We argue that a malware detection scheme which is *realtime deployable* should use an intelligent yet simple static analysis technique. In this paper we propose a framework, called *PE-Miner*, which uses novel *structural features* to efficiently detect malicious PE files. PE is a file format which is standardized by the Microsoft Windows operating systems for executables, dynamically linked libraries (DLL), and object files. We follow a threefold research methodology in our static analysis: (1) identify a set of structural features for PE files which is computable in realtime, (2) use an efficient preprocessor for removing redundancy in the features' set, and (3) select an efficient data mining algorithm for final classification. Consequently, our proposed framework consists of three modules: the feature extraction module, the feature selection/preprocessing module, and the detection module.

We have evaluated our proposed detection framework on two independently collected malware datasets with different statistics. The first malware dataset is the VX Heavens Virus collection consisting of more than ten thousand malicious PE files [27]. The second malware dataset is the Malfease dataset, which contains more than five thousand malicious PE files [21]. We also collected more than one thousand benign PE files from our virology lab, which we use in conjunction with both malware datasets in our study. The results of our experiments

---

[1] We define a technique as *realtime deployable* if it has three properties: (1) a *tp* rate (or true positive rate) of approximately 1, (2) an *fp* rate (or false positive rate) of approximately 0, and (3) the file scanning time is comparable to existing COTS AV.

show that our PE-miner framework achieves more than 99% detection rate with less than 0.5% false alarm rate for distinguishing between benign and malicious executables. Further, our framework takes on the average only 0.244 seconds to scan a given PE file. Therefore, we can conclude that PE-Miner is *realtime deployable*, and consequently it can be easily integrated into existing COTS AV products. PE-Miner framework can also categorize the malicious executables as a function of their payload. This analysis is of great value for system administrators and malware forensic experts. An interested reader can find details in the accompanying technical report [23].

We have also compared PE-Miner with other promising malware detection schemes proposed by Perdisci et al. [18], Schultz et al. [22], and Kolter et al. [11]. These techniques use some variation of n-gram analysis for malware detection. PE-Miner provides better detection accuracy[2] with significantly smaller processing overheads compared with these approaches. We believe that the superior performance of PE-Miner is attributable to a rich set of novel PE format specific structural features, which provides relevant information for better detection accuracy [10]. In comparison, n-gram based techniques are more suitable for classification of loosely structured data; therefore, they fail to exploit format specific structural information of a PE file. As a result, they provide lower detection rates and have higher processing overheads as compared to PE-Miner. Our experiments also demonstrate that the detection mechanism of PE-Miner does not show any significant bias towards packed/non-packed PE files. Finally, we investigate the robustness of PE-Miner against "crafty" attacks which are specifically designed to evade detection mechanism of PE-Miner. Our results show that PE-Miner is resilient to majority of such evasion attacks.

2 PE-Miner Framework

In this section, we discuss our proposed PE-Miner framework. We set the following strict requirements on our PE-Miner framework to ensure that our research is enacted with a product development cycle that has a short time-to-market:

- It must be a pure non-signature based framework with an ability to detect zero-day malicious PE files.
- It must be *realtime deployable*. To this end, we say that it should have more than 99% *tp* rate and less than 1% *fp* rate. We argue that it is still a challenge for non-signature based techniques to achieve these true and false positive rates. Moreover, its time to scan a PE file must be comparable to those of existing COTS AV products.

[2] Throughout this text, the terms *detection accuracy* and *Area Under ROC Curve (AUC)* are used interchangeably. ROC curves are extensively used in machine learning and data mining to depict the tradeoff between the true positive rate and false positive rate of a classifier. The AUC ($0 \leq \text{AUC} \leq 1$) is used as a yardstick to determine the detection accuracy from ROC curve. Higher values of AUC mean high *tp* rate and low *fp* rate [28]. At AUC = 1, *tp* rate = 1 and *fp* rate = 0.

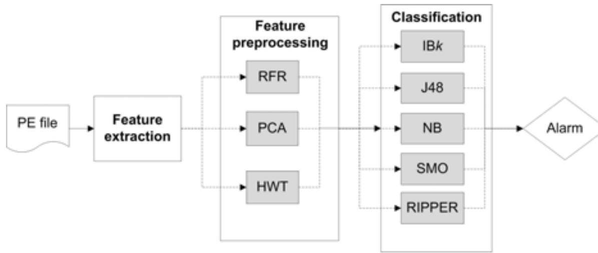


Fig. 1. The architecture of our PE-Miner framework

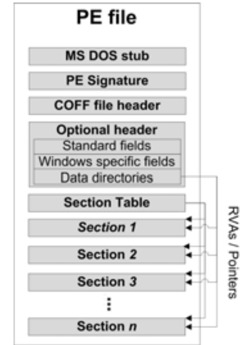


Fig. 2. The PE file format

– Its design must be modular that allows for the plug-n-play design philosophy. This feature will be useful in customizing the detection framework to specific requirements, such as porting it to the file formats used by other operating systems.

We have evolved the final modular architecture of our PE-Miner framework in a question oriented engineering fashion. In our research, we systematically raised following relevant questions, analyzed their potential solutions, and finally selected the best one through extensive empirical studies.

1. Which PE format specific features can be statically extracted from PE files to distinguish between benign and malicious files? Moreover, are the format specific features better than the existing n-grams or string-based features in terms of detection accuracy and efficiency?
2. Do we need to deploy preprocessors on the features' set? If yes then which preprocessors are best suited for the raw features' set?
3. Which are the best back-end classification algorithms in terms of detection accuracy and processing overheads.

Our PE-Miner framework consists of three main modules inline with the above-mentioned vision: (1) feature extraction, (2) feature preprocessing, and (3) classification (see Figure 1). We now discuss each module separately.

2.1 Feature Extraction

Let us revisit the PE file format [12] before we start discussing the structural features used in our features' set. A PE file consists of a PE file header, a section table (section headers) followed by the sections' data. The PE file header consists of a MS DOS stub, a PE file signature, a COFF (Common Object File Format) header, and an optional header. It contains important information about a file

Table 1. List of the features extracted from PE files

Feature Description	Type	Quantity
DLLs referred	binary	73
COFF file header	integer	7
Optional header – standard fields	integer	9
Optional header – Windows specific fields	integer	22
Optional header – data directories	integer	30
.text section – header fields	integer	9
.data section – header fields	integer	9
.rsrc section – header fields	integer	9
Resource directory table & resources	integer	21
Total		189

such as the number of sections, the size of the stack and the heap, etc. The section table contains important information about the sections that follow it, such as their name, offset and size. These sections contain the actual data such as code, initialized data, exports, imports and resources [12], [15].

Figure 2 shows an overview of the PE file format [12], [15]. It is important to note that the section table contains Relative Virtual Addresses (RVAs) and the pointers to the start of every section. On the other hand, the data directories in an optional header contain references to various tables (such as import, export, resource, etc.) present in different sections. These references, if appropriately analyzed, can provide useful information.

We believe that this structural information about a PE file should be leveraged to extract features that have the potential to achieve high detection accuracy. Using this principle, we statically extract a set of large number of features from a given PE file[3]. These features are summarized in Table 1. In the discussion below, we first intuitively argue about the features that have the potential to distinguish between benign and malicious files. We then show interesting observations derived from the executable datasets used in our empirical studies.

DLLs referred. The list of DLLs referred in an executable effectively provides an overview of its functionality. For example, if an executable calls `WINSOCK.DLL` or `WSOCK.DLL` then it is expected to perform network related activities. However, there can be exceptions to this assumption as well. In [22], Schultz et al. have used the conjunction of DLL names, with a similar functionality, as binary features. The results of their experiments show that this feature helps to attain reasonable detection accuracy. However, our pilot experimental studies have revealed that using them as individual binary features can reveal more information, and hence can be more helpful in detecting malicious PE files. In this study, we have used 73 core functionality DLLs as features. Their list and functionality is detailed in [23]. Table 2 shows the mean feature values for the two DLLs[4]. Interestingly, `WSOCK32.DLL` and `WININET.DLL` are used by the majority of backdoors, nukers, flooders, hacktools, worms, and trojans to access the resources on the network

[3] A well-known Microsoft Visual C++ utility, called **dumpbin**, dumps the relevant information which is present inside a given PE file [4]. Another freely available utility, called **pedump**, also does the required task [20].

[4] The details of the datasets and their categorization are available in Section 3.

Table 2. Mean values of the extracted features. The bold values in every row highlight interesting outliers.

Dataset	VX Heavens									Malfease
Name of Feature	Benign	Backdoor + Sniffer	Constructor + Virtool	DoS + Nuker	Flooder	Exploit + Hacktool	Worm	Trojan	Virus	-
WSOCK32.DLL	0.037	**0.503**	0.038	**0.188**	**0.353**	**0.261**	**0.562**	**0.242**	0.053	0.065
WININET.DLL	0.073	**0.132**	0.009	0.013	0.04	**0.141**	0.004	**0.103**	0.019	0.086
# Symbols	**430.2**	2.0*E*6	14.7	59.4	25.8	3.5*E*6	38.8	4.1*E*6	1.0*E*6	2.7*E*7
Maj Linker Ver	**4.7**	14.4	11.2	14.1	12.1	12.3	18.7	12.2	19.3	6.5
Init Data Size (*E*5)	**4.4**	1.1	0.5	0.4	0.8	0.7	0.4	0.4	0.1	0.6
Maj Img Ver	**163.1**	1.6	6.3	0.4	0.6	11.2	0.3	6.0	53.6	0.2
DLL Char	**5.8x10^3**	0.0	0.0	0.0	0.0	24.9	0.0	3.1	230.8	18.7
Exp Tbl Size (*E*2)	**13.7**	2.4	1.7	**14.1**	5.0	0.3	1.2	2.1	0.9	0.05
Imp Tbl Size (*E*2)	5.8	**19.2**	6.1	7.9	**20.8**	7.1	**23.4**	**10.3**	6.2	4.7
Rsrc Tbl Size (*E*4)	**32.6**	5.5	1.5	1.4	6.2	1.0	2.6	2.2	0.5	5.9
Except Tbl Size	12.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**3.5**
.data Raw Size (*E*3)	**25.2**	8.4	5.6	6.3	6.0	7.9	6.1	5.5	6.7	22.1
# Cursors	**14.5**	6.4	6.7	7.4	6.1	5.9	5.8	6.0	3.0	6.8
# Bitmaps	**12.6**	1.2	0.0	1.0	0.6	0.7	1.2	1.4	2.4	0.5
# Icons	**17.6**	2.5	1.9	2.7	2.0	2.1	1.8	1.9	4.5	2.2
# Dialogs	**10.9**	3.2	1.5	3.2	1.5	2.0	1.9	1.7	2.2	2.3
# Group Cursors	**11.6**	6.0	6.6	7.2	5.8	5.8	5.4	5.7	2.7	6.7
# Group Icons	**4.1**	1.0	0.7	1.0	0.8	0.7	0.5	0.7	1.5	0.6

and the Internet. Therefore, the applications misusing these DLLs might provide a strong indication of a possible covert network activity.

COFF file header. The COFF file header contains important information such as the type of the machine for which the file is intended, the nature of the file (DLL, EXE, or OBJ etc.), the number of sections, and the number of symbols. It is interesting to note in Table 2 that a reasonable number of symbols are present in benign executables. The malicious executables, however, either contain too many or too few symbols.

Optional header: standard fields. The interesting information in the standard fields of the optional header include the linker version used to create an executable, the size of the code, the size of the initialized data, the size of the uninitialized data, and the address of the entry point. Table 2 shows that the values of major linker version and the size of the initialized data have a significant difference in the benign and malicious executables. The size of the initialized data in benign executables is usually significantly higher compared to those of the malicious executables.

Optional header: Windows specific fields. The Windows specific fields of the optional header include information about the operating system version, the image version, the checksum, the size of the stack and the heap. It can be seen in Table 2 that the values of fields such as the major image version and the DLL characteristics are usually set to zero in the malicious executables. In comparison, their values are significantly higher in the benign executables.

Optional header: data directories. The data directories of the optional header provide pointers to the actual data present in the sections following it. It includes the information about export, import, resource, exception, debug, certificate, and base relocation tables. Therefore, it effectively provides a summary of the contents of an executable. Table 2 highlights that the size of the export

table is higher for the benign executables and nukers as compared to those of other malicious executables. Another interesting observation in Table 2 is that the backdoors, flooders, worms and trojans mostly have a bigger import table size. It can be intuitively argued that they usually import network functionalities which increase the size of their import table. The size of the resource table, on the other hand, is higher for the benign executables as compared to those of the malicious executables. The exception table is mostly absent in the malicious executables.

Section headers. The section headers provide important characteristics of a section such as its address, size, number of relocations, and line numbers. In this study, we have only considered text, data and resource sections because they are commonly present in the executables. Note that the size of the data section (if present) is relatively larger for the benign executables.

Resource directory table & resources. The resource directory table provides an overview of the resources that are present in the resource section of an executable file. We consider the actual count of various types of resources that are present in the resource section of an executable file. The typical examples of resources include cursors, bitmaps, icons, menus, dialogs, fonts, group cursors, and user defined resources. Intuitively and as shown in Table 2, the number of these resources is relatively higher for the benign executables.

2.2 Feature Selection/Preprocessing

We have now identified our features' set that consists of a number of statically computable features – 189 to be precise – based on the structural information of the PE files. It is possible that some of the features might not convey useful information in a particular scenario. Therefore, it makes sense to remove or combine them with other similar features to reduce the dimensionality of our input feature space. Moreover, this preprocessing on the raw extracted features' set also reduces the processing overheads in training and testing of classifiers, and can possibly also improve the detection accuracy of classifiers. In this study, we have used three well-known features' selection/preprocessing filters. We provide their short descriptions in the following text. More details can be found in [29].

Redundant Feature Removal (RFR). We apply this filter to remove those features that do not vary at all or show significantly large variation i.e. they have approximately uniform-random behavior. Consequently, this filter removes all features that have either constant values or show a variance above a threshold or both.

Principal Component Analysis (PCA). The Principal Component Analysis (PCA) is a well-known filter for dimensionality reduction. It is especially useful when the input data has high dimensionality – sometimes referred to as *curse of dimensionality*. This dimensionality reduction can possibly improve the quality of an analysis on a given data if the dataset consists of highly correlated or

redundant features. However, this dimensionality reduction may result in information loss (i.e. reduction in data variance) as well. One has to carefully choose the appropriate balance for this tradeoff. We apply PCA filter to remove/combine correlated features for dimensionality reduction.

Haar Wavelet Transform (HWT). The principle of this technique is that the most relevant information is stored with the highest coefficients at each order of a transform. The lower order coefficients can be ignored to get only the most relevant information. The wavelet transform has also been used for dimensionality reduction. The wavelet transform technique has been extensively used in the image compression but is never evaluated in the malware detection domain. The Haar wavelet is one of the simplest wavelets and is known to provide reasonable accuracy. The application of Haar wavelet transform requires input data to be normalized. Therefore, we have passed the data through a *normalize* filter before applying HWT.

2.3 Classification

Once the dimensionality of the input features' set is reduced by applying one of the above-mentioned preprocessing filters, it is given as an input to the well-known data mining algorithms for classification. In this study we have used five classifiers: (1) instance based learner (IBk), (2) decision tree (J48), (3) Naïve Bayes (NB), (4) inductive rule learner (RIPPER), and (5) support vector machines using sequential minimal optimization (SMO). An interested reader can find their details in the accompanying technical report [23].

3 Datasets

In this section, we present an overview of the datasets used in our study. We have collected 1,447 benign PE files from the local network of our virology lab. The collection contains executables such as Packet CAPture (PCAP) file parsers compiled by MS Visual Studio 6.0, compressed installation executables, and MS Windows XP/Vista applications' executables. The diversity of the benign files is also evident from their sizes, which range from a minimum of 4 KB to a maximum of 104,588 KB (see Table 3).

Moreover, we have used two malware collections in our study. First is the VX Heavens Virus Collection, which is *labeled* and is publicly available for free download [27]. We only consider PE files to maintain focus. Our filtered dataset contains 10,339 malicious PE files. The second dataset is the Malfease malware dataset [21], which consists of 5,586 *unlabeled* malicious PE files.

In order to conduct a comprehensive study, we further categorize the malicious PE files as a function of their payload[5]. The malicious executables are subdivided into eight major categories such as *virus*, *trojan*, *worm*, etc [7]. Moreover, we

[5] Since the Malfease malware collection is unlabeled; therefore, it is not possible to divide it into different malware categories.

Table 3. Statistics of the data used in this study

Dataset	VX Heavens									Malfease
-	Benign	Backdoor + Sniffer	Constructor + Virtool	DoS + Nuker	Flooder	Exploit + Hacktool	Worm	Trojan	Virus	-
Quantity	1, 447	3, 455	367	267	358	243	1, 483	3, 114	1, 052	5, 586
Avg. Size (KB)	1, 263	270	234	176	298	156	72	136	50	285
Min. Size (KB)	4	1	4	3	6	4	2	1	2	1
Max. Size (KB)	104, 588	9, 277	5, 832	1, 301	14, 692	1, 924	2, 733	4, 014	1, 332	5, 746
UPX	17	786	79	15	32	43	353	622	48	470
ASPack	2	432	21	16	25	15	66	371	10	187
Misc. Packed	372	325	47	31	58	38	471	170	71	1, 909
Borland C/C++	15	56	8	15	10	6	13	63	18	11
Borland Delphi	13	589	13	65	64	8	76	379	71	342
Visual Basic	4	719	106	39	126	38	210	674	119	809
Visual C++	526	333	19	51	29	59	89	619	96	351
Visual C#	56	0	0	0	1	0	5	1	6	1
Misc. Other	9	49	9	2	3	2	4	15	7	5
Non-packed (%)	43.1	50.5	42.2	64.4	65.1	46.5	26.8	56.2	30.1	27.2
Packed (%)	27.0	44.7	40.1	23.2	32.1	39.5	60.0	37.4	12.3	46.6
Not Found (%)	29.9	4.8	17.7	12.4	2.8	14.0	13.2	6.4	57.6	26.2

have combined some categories that have similar functionality. For example, we have combined *constructor* and *virtool* to create a single *constructor + virtool* category. This unification increases the number of malware samples per category. Brief introductions of every malware category are provided in the accompanying technical report [23].

Table 3 provides the detailed statistics of the malware used in our study. It can be noted that the average size of the malicious executables is smaller than that of the benign executables. Further, some executables used in our study are encrypted and/or compressed (packed). The detailed statistics about packing are also tabulated in Table 3. We use PEiD [16] and Protection ID for detecting packed executables [19][6].

Our analysis shows that VX Heavens Virus collection contains 40.1% packed and 47.2% non-packed PE files. However, approximately 12.7% malicious PE files cannot be classified as either packed or non-packed by PEiD and Protection ID. The Malfease collection contains 46.6% packed and 27.2% non-packed malicious PE files. Similarly, 26.2% malicious PE files cannot be classified as packed or non-packed. Therefore, we can say that packed/non-packed malware distribution in the VX Heavens virus collection is relatively more balanced than the Malfease dataset. In our collection of benign files, 43.1% are packed and 27.0% are non-packed PE files respectively. Similarly, 29.9% benign files are not detected by PEiD and Protection ID. An interesting observation is that the benign PE files are mostly packed using nonstandard and custom developed packers. We speculate that a significant portion of the packed executables are not classified as packed because the signatures of their respective packers are not present in the database of PEiD or Protection ID. *Note that we do not manually unpack any PE file prior to the processing of our PE-Miner.*

[6] We acknowledge the fact that PEiD and Protection ID are signature based packer detectors and can have significant false negatives.

4 Related Work

We now briefly describe the most relevant non-signature based malware detection techniques. These techniques are proposed by Perdisci et al. [18], Schultz et al. [22] and Kolter et al. [11]. We briefly summarize their working principles in the following paragraphs but an interested reader can find their detailed description in [23].

In [18], the authors proposed McBoost that uses two classifiers – C1 and C2 – for classification of non-packed and packed PE files respectively. A custom developed unpacker is used to extract the hidden code from the packed PE files and the output of the unpacker is given as an input to the C2 classifier. Unfortunately, we could not obtain its source code or binary due to licensing related problems. Furthermore, its implementation is not within the scope of our current work. Consequently, we only evaluate the C1 module of McBoost which works only for non-packed PE files. Therefore, we acknowledge that our McBoost results should be considered only preliminary.

In [22], Schultz et al. have proposed three independent techniques for detecting malicious PE files. The first technique, uses the information about DLLs, function calls and their invocation counts. However, the authors did not provide enough information about the used DLLs and function names; therefore, it is not possible for us to implement it. But we have implemented the second approach (titled *strings*) which uses strings as binary features i.e. present or absent. The third technique uses two byte words as binary features. This technique is later improved in a seminal work by Kolter et al. [11] which uses 4-grams as binary features. Therefore, we include the technique of Kolter et al. (titled *KM*) in our comparative evaluation.

5 Experimental Results

We have compared our PE-Miner framework with recently proposed promising techniques by Perdisci et al. [18], Schultz et al. [22], and Kolter et al. [11]. We have used the standard 10 fold cross-validation process in our experiments, i.e., the dataset is randomly divided into 10 smaller subsets, where 9 subsets are used for training and 1 subset is used for testing. The process is repeated 10 times for every combination. This methodology helps in systematically evaluating the effectiveness of our approach to detect previously unknown (i.e. zero-day) malicious PE files. The ROC curves are generated by varying the threshold on output class probability [5], [28]. The AUC is used as a yardstick to determine the detection accuracy of each approach. We have done the experiments on an Intel Pentium Core 2 Duo 2.19 GHz processor with 2 GB RAM. The Microsoft Windows XP SP2 is installed on this machine.

5.1 Malicious PE File Detection

In our first experimental study, we attempt to distinguish between benign and malicious PE files. To get better insights, we have done independent experiments

Table 4. AUCs for detecting the malicious executables. The bold entries in each column represent the best results.

Dataset	VX Heavens									Malfease
Malware	Backdoor + Sniffer	Constructor + Virtool	DoS + Nuker	Flooder	Exploit + Hacktool	Worm	Trojan	Virus	Average	-
PE-Miner — RFR										
IBK	0.992	0.996	0.995	0.994	0.998	0.979	0.984	0.994	**0.992**	0.986
J48	0.993	**0.998**	0.987	0.993	0.999	0.979	**0.992**	0.993	**0.992**	0.979
NB	0.971	0.978	0.966	0.973	0.987	0.972	0.974	0.986	0.976	0.976
RIPPER	**0.996**	0.996	0.977	0.981	0.999	**0.988**	0.988	0.996	0.990	0.985
SMO	0.991	0.990	0.991	0.993	0.997	0.975	0.978	0.992	0.988	0.963
PE-Miner — PCA										
IBK	0.989	0.996	0.994	0.995	0.998	0.976	0.984	0.993	0.991	0.984
J48	0.980	0.966	0.929	0.960	0.987	0.936	0.951	0.985	0.962	0.945
NB	0.961	0.990	0.993	0.996	0.996	0.964	0.956	0.990	0.981	0.898
RIPPER	0.982	0.978	**0.996**	0.974	0.977	0.949	0.968	0.987	0.976	0.952
SMO	0.990	0.992	0.989	0.995	0.995	0.958	0.965	0.992	0.985	0.954
PE-Miner — HWT										
IBK	0.991	0.996	**0.996**	**0.998**	**1.000**	0.978	0.985	0.995	**0.992**	0.986
J48	0.995	0.997	0.993	0.988	0.997	0.978	0.991	0.999	**0.992**	0.977
NB	0.989	0.982	0.983	0.987	0.990	0.978	0.972	0.990	0.984	0.960
RIPPER	0.994	0.997	0.982	0.990	0.997	0.983	0.990	**1.000**	**0.992**	**0.987**
SMO	0.990	0.995	0.991	0.996	**1.000**	0.972	0.973	0.994	0.989	0.964
McBoost — C1 only										
IBK	0.941	0.935	0.875	0.960	0.832	0.938	0.930	0.914	0.916	0.949
J48	0.866	0.895	0.809	0.893	0.731	0.906	0.902	0.882	0.860	0.860
NB	0.831	0.924	0.723	0.889	0.795	0.873	0.886	0.844	0.846	0.817
RIPPER	0.833	0.888	0.744	0.918	0.660	0.866	0.838	0.844	0.824	0.860
SMO	0.802	0.887	0.759	0.910	0.678	0.854	0.805	0.827	0.815	0.835
Strings										
IBK	0.949	0.860	0.902	0.980	0.925	0.928	0.863	0.952	0.920	0.944
J48	0.913	0.834	0.862	0.695	0.871	0.908	0.836	0.938	0.857	0.929
NB	0.920	0.830	0.882	0.726	0.886	0.901	0.828	0.905	0.860	0.930
RIPPER	0.843	0.797	0.714	0.578	0.712	0.892	0.743	0.929	0.776	0.927
SMO	0.855	0.817	0.705	0.775	0.583	0.871	0.756	0.883	0.781	0.933
KM										
IBK	0.984	0.934	0.983	0.971	0.983	0.987	0.979	0.986	0.976	0.980
J48	0.953	0.940	0.916	0.907	0.916	0.957	0.951	0.953	0.937	0.952
NB	0.943	0.959	0.961	0.952	0.961	0.968	0.954	0.954	0.957	0.961
RIPPER	0.951	0.944	0.924	0.921	0.924	0.964	0.948	0.948	0.941	0.971
SMO	0.949	0.946	0.952	0.927	0.952	0.961	0.940	0.938	0.946	0.960

with benign and each of the eight types of the malicious executables. The five data mining algorithms, namely IBk, J48, NB, RIPPER, and SMO, are deployed on top of each approach (namely PE-Miner with RFR, PE-Miner with PCA, PE-Miner with HWT, McBoost (C1 only) by Perdisci et al. [18], strings approach by Schultz et al. [22], and KM by Kolter et al. [11]). This results in a total of 270 experimental runs each with 10-fold cross validation. We tabulate our results for this study in Table 4 and now answer different questions that we raised in Section 2 in a chronological fashion.

Which features' set is the best? Table 4 tabulates the AUCs for PE-Miner using three different preprocessing filters (RFR, PCA and HWT), McBoost, strings and KM [11]. A macro level scan through the table clearly shows the supremacy of PE-Miner based approaches with AUCs more than 0.99 for most of the malware types and even approaching 1.00 for some malware types. For PE-Miner, RFR and HWT preprocessing lead to the best average results with more than 0.99 AUC.

The strings approach gives the worst detection accuracy. The KM approach is better than the strings approach but inferior to our PE-Miner. This is expected because the string features are not stable as compiling a given piece of code by

Table 5. The processing overheads (in seconds/file) of different feature selection, extraction and preprocessing schemes

	PE-Miner (RFR)	PE-Miner (PCA)	PE-Miner (HWT)	McBoost	Strings	KM
Selection	-	-	-	2.839	5.289	31.499
Extraction	0.228	0.228	0.228	0.198	0.130	0.220
Preprocessing	0.007	0.009	0.012	-	-	-
Total	**0.235**	0.237	0.240	3.037	5.419	31.719

using different compilers leads to different sets of strings. Our analysis shows that KM approach is more resilient to variation in the string sets because it uses a combination of string and non-string features. The results obtained for KM approach (AUC= 0.95) are also consistent with the results reported in [11]. The C1 module of McBoost also provides relatively inferior detection accuracies which are as low as 0.66 for exploit+hacktool category. It is important to note that the C1 module of McBoost is functionally similar to the techniques proposed by Schultz et al. and Kolter et al. The only significant difference is that C1 operates only on the code sections of the non-packed PE files whereas the other techniques operate on complete files.

It is important to emphasize that both strings and KM approaches incur large overheads in the feature selection process (see Table 5[7]). Kolter et al. have confirmed that their implementation of information gain calculation for feature selection took almost a day for every run. To make our implementation of n-grams more efficient, we use `hash_map` STL containers in the Visual C++ [8]. Our experiments show that the feature selection process in KM still takes more than 31 seconds per file even with our optimized implementation. The optimized strings approach takes, on the average, more than 5 seconds per file for feature selection. The optimized McBoost (C1 only) approach takes an average of more than 2 seconds per file for feature selection[8]. These approaches have processing overheads because the time to calculate information gain increases exponentially with the number of unique n-grams (or strings). On the other hand, PE-Miner does not suffer from such serious bottlenecks. The application of RFR, PCA, or HWT filters takes only about a hundredth of a second.

Which classification algorithm is the best? We can conclude from Table 4 that J48 outperforms the rest of the data mining classifiers in terms of the detection accuracy in most of the cases. Moreover, Table 6 shows that J48 has one of the smallest processing overheads both in training and testing. RIPPER and IBk closely follow the detection accuracy of J48. However, they are infeasible for realtime deployment because of the high processing overheads in the training and the testing phases respectively. The processing overheads of training RIPPER are the highest among all classifiers. In comparison, IBk does not require a training phase but its processing overheads in the testing phase are the highest. Further,

[7] The results in Table 5 are averaged over 100 runs.

[8] Note that the complete McBoost system also uses unpacker for extraction of hidden code. This process is time consuming as reported by the authors in [18].

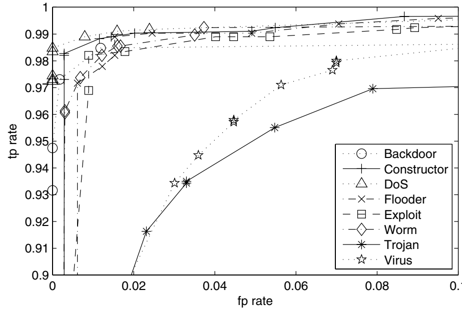


Fig. 3. The magnified ROC plots for detecting the malicious executables using PE-Miner utilizing J48 preprocessed with RFR filter

Table 6. The processing overheads (in seconds/file) of different features and classification algorithms

	Training					Testing				
	IBK	J48	NB	RIPPER	SMO	IBK	J48	NB	RIPPER	SMO
PE-Miner (RFR)	-	0.008	0.001	0.269	0.199	0.032	0.001	0.002	0.002	0.002
PE-Miner (PCA)	-	0.007	0.001	0.264	0.179	0.035	0.001	0.001	0.001	0.002
PE-Miner (HWT)	-	0.007	0.001	0.252	0.147	0.032	0.001	0.002	0.001	0.002
McBoost	-	0.021	0.004	1.305	1.122	0.218	0.010	0.007	0.005	0.022
Strings	-	0.009	0.002	0.799	0.838	0.163	0.003	0.003	0.002	0.003
KM	-	0.024	0.004	1.510	1.018	0.254	0.018	0.007	0.005	0.020

Naïve Bayes gives the worst detection accuracy because it assumes independence among input features. Intuitively speaking, this assumption does not hold for the features' sets used in our study. Note that Naïve Bayes has very small learning and testing overheads (see Table 6[9]).

Which malware category is the most challenging to detect? An overview of Table 4 suggests that the most challenging malware categories are worms and trojans. The average AUC values of the compared techniques for worms and trojans are approximately 0.95. The poor detection accuracy is attributed to the fact that the trojans are inherently designed to appear similar to the benign executables. Therefore, it is a difficult challenge to distinguish between trojans and benign PE files. Our PE-Miner still achieves on the average 0.98 AUC for worms and trojans which is quite reasonable. Figure 3 shows that for other malware categories, PE-Miner (with RFR preprocessor) has AUCs more than 0.99.

5.2 Miscellaneous Discussions

We conclude our comparative study with an answer to an important issue: *which of the compared techniques meet the criterion of being realtime deployable?* (see Section 2). We tabulate the AUC and the scan time of the best techniques in

[9] The results in Table 6 are averaged over 100 runs.

Table 7. Realtime deployable analysis of the best techniques

Technique	Classifier	AUC	Scan Time (sec/file)	Is Realtime Deployable?
PE-Miner (RFR)	J48	0.991	0.244	"Yes"
McBoost	IBk	0.926	3.255	No
Strings	IBk	0.927	5.582	No
KM	IBk	0.977	31.973	No
AVG Free 8.0 [1]	-	-	0.159	-
Panda 7.01 [14]	-	-	0.131	-

Table 8. Portion of the developed decision trees for distinguishing between benign and backdoor+sniffer

```
NumMessageTable <= 0
|    SizeLoadConfigTable <= 0
|    |    TimeDateStamp <= 1000000000
|    |     |       NumCursor <= 1
|    |     |        |     NumAccelerators <= 0
|    |     |        |      |       NumBitmap <= 0: malicious
|    |     |        |      |       NumBitmap > 0: benign
|    |     |        |     NumAccelerators > 0:malicious
|    |     |       NumCursor > 1:malicious
```

Table 7. Moreover, we also show the scan time of two well-known COTS AV products for doing the *realtime deployable* analysis of different non-signature based techniques. It is clear that PE-Miner (RFR) with J48 classifier is the only non-signature based technique that satisfies the criterion of being *realtime deployable*. One might argue that PE-Miner framework provides only a small improvement in detection accuracy over the KM approach. *But then KM has the worst scan time of* 31.97 *seconds per file (see Table 7)*. It is very important to interpret the results in Table 7 from a security expert's perspective. For example, if a malware detector scans ten thousand files with an AUC of 0.97, it will not detect approximately 300 malicious files. In comparison, a detector with an AUC of 0.99 will miss only 100 files, which is a 66.6% improvement in the number of missed files [2]. Therefore, we argue that from a security expert's perspective, even a small improvement in the detection accuracy is significant in the limiting case when the detection accuracy approaches to 1.00.

An additional benefit of PE-Miner is that it provides insights about the learning models of different classifiers that can be of great value to malware forensic experts. We show a partial subtree of J48 for categorizing benign and malicious PE files in Table 8. The message tables mostly do not exist in the backdoor+sniffer categories. The TimeDateStamp is usually obfuscated in the malicious executables. The number of resources are generally smaller in malicious PE files, whereas the benign files tend to have larger number of resources such as menus, icons, and user defined resources. Similar insights are also provided by the rules developed in the training phase of RIPPER.

In [9], the authors have pointed out several difficulties in parsing PE files. In our experiments, we have also observed various anomalies in parsing the structure of malicious PE files. Table 9 contains the list of anomalies which we

Table 9. List of the anomalies observed in parsing malicious PE files

ID	Description
1	Large number of sections
2	SizeOfHeader field is unaligned
3	Overlapping DoS and PE headers
4	Large virtual size in a section
5	Large raw data size in a section
6	Zero/Non-zero pair in data directory table
7	Large pointer in data directory entry
8	Size of section is too large
9	Section name garbled (non printable characters)
10	There is an unknown overlay region
11	Out of file pointer

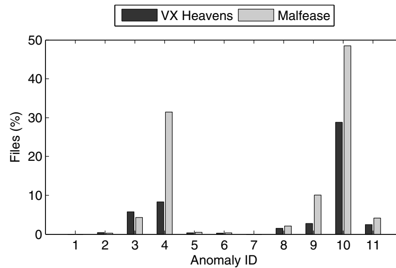


Fig. 4. Statistics of anomalies observed in parsing malicious PE files

have observed in parsing malicious PE files. A significant proportion of malicious PE files have anomalous structure which can crash a naïve PE file parser. Figure 4 provides the statistics of anomalies which we have observed in parsing malicious PE files of VX Heavens and Malfease collections. To this end, we have developed a set of heuristics which successfully handle the above-mentioned anomalies.

6 Robustness and Reliability Analysis of PE-Miner

We have now established the fact that *PE-Miner is a realtime deployable scheme for zero-day malware detection.* A careful reader might ask whether the statement still holds if the "ground truth" is now changed as: (1) we cannot trust the classification of signature based packer detectors PEiD and Protection ID, and (2) a "crafty" attacker can forge the features of malicious files with those of benign files to evade detection. In this section, we do a stress and regression testing of PE-Miner to analyze robustness of its features and its resilience to potential evasive techniques.

6.1 Robustness Analysis of Extracted Features

It is a well-known fact that signature based packer detector PEiD, which we are using to distinguish between packed and non-packed executables, has approximately 30% false negative rate [17]. In order to convince ourselves that our

extracted features are actually "robust", we evaluate PE-Miner in four scenarios: (1) training PE-Miner on 70% non-packed PE files and 30% packed PE files and testing on the remaining 70% packed PE files, (2) training PE-Miner on non-packed PE files only and testing on packed PE files, (3) training PE-Miner on packed PE files only and then testing on non-packed PE files, and (4) testing PE-Miner on a "difficult" dataset that consists of packed benign and non-packed malicious PE files. We assert that the scenarios (2) and (3) – although unrealistic – still provide valuable insight into the extent of bias, that PE-Miner might have, towards detection of packed/non-packed executables.

We want to emphasize an important point that there is no confusion about "ground truth" for packed executables in above-mentioned four scenarios because a packer only detects a file as "packed" if it has its signature in its database. The confusion about "ground truth", however, stems in the fact that a reasonable proportion of packed PE files could be misclassified as non-packed because of false negative rate of PEiD. Note that the false negatives of PEiD, reported in [17], consist of two types: (1) packed PE files that are misclassified as non-packed, and (2) PE files that are unclassified. We have not included unclassified files in our dataset to remove the false negatives of the second type.

Scenario 1: Detection of packed benign and malicious PE files. The motivation behind the first scenario is to test if PE-Miner can distinguish between packed benign and packed malware, regardless of the type of packer. In order to ensure that our features are not influenced by the type of packing tool used to encrypt PE files, our "packed-only" dataset contains PE files (both benign and malware) packed using a variety of packers like UPX, ASPack, Armadillo, PECompact, WWPack32, Virogen Crypt 0.75, UPS-Scrambler, PEBundle and PEPack etc. Moreover, the "packed-only" dataset contains on the average 44% and 56% packed malicious and benign PE files respectively. We train PE-Miner on 70% non-packed executables and 30% packed executables and then test it on the remaining 70% packed executables. The results of our experiments for this scenario are tabulated in Table 10. We can easily conclude that PE-Miner has shown good resilience in terms of detecting accuracy once it is tested on packed benign and malicious PE files from both datasets.

Scenarios 2 and 3: Detection of packed/non-packed malicious PE files. In the second experiment, we train PE-Miner on non-packed benign and malicious PE files and test it on packed benign and malicious PE files. Note that this scenario is more challenging because the training dataset contains significantly less number of packed files compared with the first scenario. In the third experiment, we train PE-Miner on packed benign and malicious PE files and test on non-packed benign and malicious PE files. The results of these experiments are tabulated in Table 10. It is clear from Table 10 that the detection accuracy of PE-Miner (RFR-J48) drops to 0.96, when it is trained on non-packed executables and tested on the packed executables. Likewise, the average detection accuracy of PE-Miner (RFR-J48) drops to 0.90 for the third scenario. Remember once we train PE-Miner on "packed only" dataset, then it gets 0% exposure

Table 10. An analysis of robustness of extracted features of PE-Miner (RFR) in different scenarios

Dataset	VX Heavens									Malfease
Malware	Backdoor + Sniffer	Constructor + Virtool	DoS + Nuker	Flooder	Exploit + Hacktool	Worm	Trojan	Virus	Average	-
Scenario 1: Detection of packed benign and malicious PE files										
IBK	0.999	1.000	1.000	0.999	0.999	0.998	0.999	0.999	0.999	0.812
J48	0.996	1.000	1.000	0.999	0.999	0.998	0.993	0.999	**0.998**	**0.991**
NB	0.971	0.988	0.963	0.955	0.996	0.980	0.978	0.987	0.977	0.934
RIPPER	0.997	0.996	0.999	0.990	0.993	0.985	0.858	0.998	0.977	0.988
SMO	0.985	0.998	1.000	0.996	0.994	0.994	0.985	0.998	0.994	0.706
Scenario 2: Training using non-packed executables only and testing using packed executables										
IBK	0.986	0.965	0.912	0.963	0.998	0.993	0.850	0.989	0.957	0.917
J48	0.982	0.999	0.998	0.937	0.999	0.963	0.857	0.954	**0.961**	**0.968**
NB	0.927	0.899	0.842	0.809	0.966	0.911	0.857	0.965	0.897	0.780
RIPPER	0.989	0.995	0.998	0.995	0.986	0.962	0.858	0.853	0.954	0.937
SMO	0.983	0.772	0.905	0.691	0.996	0.737	0.651	0.852	0.823	0.859
Scenario 3: Training using packed executables only and testing using non-packed executables										
IBK	0.975	0.965	0.964	0.878	0.793	0.982	0.911	0.904	0.921	0.855
J48	0.951	0.908	0.919	0.940	0.726	0.958	0.903	0.881	**0.898**	**0.903**
NB	0.685	0.965	0.668	0.633	0.689	0.979	0.688	0.688	0.749	0.789
RIPPER	0.979	0.938	0.967	0.972	0.747	0.768	0.840	0.867	0.885	0.904
SMO	0.977	0.941	0.877	0.882	0.536	0.983	0.835	0.904	0.867	0.849
Scenario 4: Detection of packed benign and non-packed malicious PE files ("difficult" dataset)										
IBK	0.999	1.000	1.000	0.999	0.998	0.998	0.998	0.994	0.998	0.992
J48	0.997	0.986	0.999	0.999	0.999	0.999	0.989	0.993	**0.995**	**0.996**
NB	0.954	0.963	0.995	0.988	0.966	0.990	0.975	0.986	0.977	0.948
RIPPER	0.998	0.984	0.998	0.993	0.986	0.999	0.992	0.996	0.993	0.948
SMO	0.989	0.996	1.000	0.997	0.996	0.997	0.984	0.992	0.994	0.945

to non-packed files and this explains deterioration in the detection accuracy of PE-Miner. We conclude that the detection accuracy of PE-Miner, even in these unrealistic stress testing scenarios, gracefully degrades.

Scenario 4: Detection of packed benign and non-packed malicious PE files. In [18], the authors report an interesting study about the ability of different schemes to detect packed/non-packed executables. They show that the detection accuracy of KM approach degrades on a "difficult" dataset consisting of packed benign and non-packed malicious PE files. According to the authors in [18], KM shows a bias towards detecting packed PE files as malware and non-packed PE files as benign. We also – in line with this strategy – tested PE-Miner on a "difficult" dataset created from both malware collections used in our study. The results are tabulated in Table 10. It is important to highlight that for these experiments PE-Miner is trained on the original datasets but is tested on the "difficult" versions of both datasets. One can conclude from the results in Table 10 that PE-Miner does not show any bias towards detecting packed executables as malicious and non-packed executables as benign.

Our experiments conclude that *the extracted features are actually "robust", and as a result, PE-Miner doest not show any significant bias towards detection of packed/non-packed executables.*

6.2 Reliability of PE-Miner

Now we test PE-Miner on a "crafty" malware dataset, especially designed to circumvent detection by PE-Miner. We particularly focus our attention on the *false*

Table 11. False negative rate for detecting malicious executables with PE-Miner on the "crafty" datasets

Dataset	VX Heavens									Malfease
Malware	Backdoor + Sniffer	Constructor + Virtool	DoS + Nuker	Flooder	Exploit + Hacktool	Worm	Trojan	Virus	Average	-
# Forged Features	False negative rate									
0/189	0.001	0.000	0.000	0.000	0.004	0.000	0.004	0.007	0.002	0.001
5/189	0.002	0.000	0.000	0.000	0.004	0.000	0.004	0.007	0.002	0.001
10/189	0.002	0.000	0.000	0.000	0.004	0.011	0.004	0.014	0.004	0.004
30/189	0.002	0.003	0.000	0.012	0.023	0.011	0.011	0.014	0.009	0.004
50/189	0.002	0.003	0.000	0.012	0.023	0.016	0.011	0.014	0.010	0.004
100/189	0.096	0.003	0.000	0.012	0.023	0.050	0.445	0.176	0.101	0.004
150/189	0.658	0.003	0.000	0.583	0.795	0.611	0.558	0.221	0.429	0.426
189/189	0.996	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.999	0.998

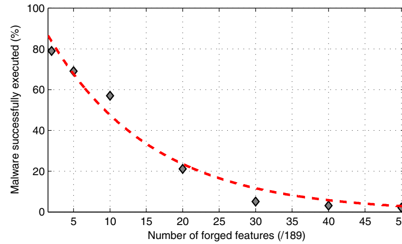


Fig. 5. Execution analysis of crafted malware files

negative rate (or miss detection rate)[10] of PE-Miner when we replace features in malicious files with those of benign files. It can be argued that if adversaries exactly know our detection methodology, they might be able to design strategies that evade detection by PE-Miner. The examples of such strategies could be especially crafted packing techniques, insertion of dummy resources, obfuscation of address pointers, and other information present in headers etc.

We have conducted an empirical study to analyze the robustness of PE-Miner to such evasive techniques. To this end, we have "crafted" malware files in the datasets to contain benign-like features. Specifically, we have created seven "crafty" datasets in which for every malware file 5, 10, 30, 50, 100, 150 and 189 random features – out of 189 features – are *forged* with the respective features from a randomly chosen benign file. We now analyze the false negative rate of PE-Miner (RFR-J48) across these "crafty" datasets. The results tabulated in Table 11 highlight the robustness of PE-Miner to such crafty attacks. The false negative rate of PE-Miner stays below 1% when fifty features are simultaneously forged. For both datasets, the average false negative rate is approximately 5% even when 100 out of 189 features are forged. This shows that a large set of features, which cover structural information of almost all portions of a PE file, used by PE-Miner make it very difficult for an attacker to evade detection – even when it manipulates majority of them at the same time.

[10] The false negative rate is defined by the fraction of malicious files wrongly classified as benign.

It should be emphasized that simultaneous manipulation of all features of a PE malware file requires significant level of skill, in-depth knowledge about the structure of a PE file, and detailed understanding of our detection framework. If an attacker tries to randomly forge, using brute-force, the structural features of a PE malware file with those of a benign PE file then he/she will inevitably end up corrupting the executable image. Consequently, the file will not load successfully into memory. We have manually executed the "crafted" malicious executables. The objective is to understand that how many features a "crafty" attacker can successfully forge without ending up corrupting the executable image. The results of our experiments are shown in Figure 5. This figure proves our hypothesis that the probability of having valid PE files decreases exponentially with an increase in the number of simultaneously forged features. In fact, the successful execution probability approaches to zero as the number of simultaneously forged features approaches to 50. Referring back to Table 11, the average false negative rate of PE-Miner is less than 1% when 50 features are simultaneously forged. Therefore, we argue that it is not a cinch for an attacker to alter malicious PE files to circumvent detection by PE-Miner. However, we accept that an attacker can evade the detection capability of PE-Miner if: (1) he/she knows the exact details of our detection framework – including the detection rules, and (2) also has the "craft" to simultaneously manipulate more than 100 structural features without corrupting the executable image.

7 Conclusion

In this paper we present, PE-Miner, a framework for detection of malicious PE files. PE-Miner leverages the structural information of PE files and the data mining algorithms to provide high detection accuracy with low processing overheads. Our implementation of PE-Miner completes a single-pass scan of all executables in the dataset (more than 17 thousand) in less than one hour. Therefore it meets all of our requirements mentioned in Section 2.

We believe that our PE-Miner framework can be ported to Unix and other non-Windows operating systems. To this end, we have identified similar structural features for the ELF file format in Unix and Unix-like operating systems. Our initial results are promising and show that PE-Miner framework is scalable across different operating systems. This dimension of our work will be the subject of forthcoming publications. Moreover, PE-Miner framework is also ideally suited for detecting malicious PE files on resource constrained mobile phones (running mobile variants of Windows) because of its small processing overheads. Finally, we are also doing research to develop techniques to fully remove the bias of PE-Miner in detecting packed/non-packed executables [24].

Acknowledgments

This work is supported in part by the National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan. The information, data, com-

ments, and views detailed herein may not necessarily reflect the endorsements of views of the National ICT R&D Fund.

We are thankful to Muhammad Umer for designing experiments to collect statistics of anomalies observed in parsing malicious PE files. We also acknowledge Marcus A. Maloof and Jeremy Z. Kolter for continuous feedbacks regarding the implementation of byte sequence approach and the experimental setup. We thank Roberto Perdisci for providing implementation details of McBoost, sharing Malfease malware dataset, and the results of their custom developed unpacker. We also thank VX Heavens moderators for making a huge malware collection publicly available and sharing packing statistics of malware. We also thank Guofei Gu and Syed Ali Khayam for providing useful feedback on an initial draft of this paper.

References

1. AVG Free Antivirus, http://free.avg.com/.
2. Axelsson, S.: The base-rate fallacy and its implications for the difficulty of intrusion detection. In: ACM Conference on Computer and Communications Security (CCS), Singapore, pp. 1–7 (1999)
3. Cheng, J., Wong, S.H.Y., Yang, H., Lu, S.: SmartSiren: virus detection and alert for smartphones. In: International Conference on Mobile Systems, Applications and Services (MobiSys), USA, pp. 258–271 (2007)
4. DUMPBIN utility, Article ID 177429, Revision 4.0, Micorsoft Help and Support (2005)
5. Fawcett, T.: ROC Graphs: Notes and Practical Considerations for Researchers, TR HPL-2003-4, HP Labs, USA (2004)
6. F-Secure Corporation, F-Secure Reports Amount of Malware Grew by 100% during 2007, Press release (2007)
7. F-Secure Virus Description Database, http://www.f-secure.com/v-descs/
8. hash_map, Visual C++ Standard Library, http://msdn.microsoft.com/en-us/library/6x7w9f6z.aspx
9. Hnatiw, N., Robinson, T., Sheehan, C., Suan, N.: PIMP MY PE: Parsing Malicious and Malformed Executables. In: Virus Bulletin Conference (VB), Austria (2007)
10. Kendall, K., McMillan, C.: Practical Malware Analysis. In: Black Hat Conference, USA (2007)
11. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: ACM International Conference on Knowledge Discovery and Data Mining (KDD), USA, pp. 470–478 (2004)
12. Microsoft Portable Executable and Common Object File Format Specification, Windows Hardware Developer Central, Updated March 2008 (2008), http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.
13. Munro, J.: Antivirus Research and Detection Techniques, Antivirus Research and Detection Techniques, Extreme Tech. (2002), http://www.extremetech.com/article2/0,2845,367051,00.asp
14. Panda Antivirus, http://www.pandasecurity.com/
15. PE file format, Webster Technical Documentation, http://webster.cs.ucr.edu/Page_TechDocs/pe.txt
16. PEiD, http://www.peid.info/

17. Perdisci, R., Lanzi, A., Lee, W.: Classification of Packed Executables for Accurate Computer Virus Detection. Elsevier Pattern Recognition Letters 29(14), 1941–1946 (2008)
18. Perdisci, R., Lanzi, A., Lee, W.: McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables. In: Annual Computer Security Applications Conference (ACSAC), pp. 301–310. IEEE Press, USA (2008)
19. Protection ID - the ultimate Protection Scanner, http://pid.gamecopyworld.com/
20. Pietrek, M.: An In-Depth Look into the Win32 Portable Executable File Format, Part 2. MSDN Magazine (March 2002)
21. Project Malfease, http://malfease.oarci.net/
22. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: IEEE Symposium on Security and Privacy (S&P), USA, pp. 38–49 (2001)
23. Shafiq, M.Z., Tabish, S.M., Mirza, F., Farooq, M.: A Framework for Efficient Mining of Structural Information to Detect Zero-Day Malicious Portable Executables, Technical Report, TR-nexGINRC-2009-21 (January 2009), http://www.nexginrc.org/papers/tr21-zubair.pdf
24. Shafiq, M.Z., Tabish, S.M., Farooq, M.: PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. In: Virus Bulletin Conference (VB), Switzerland (2009)
25. Symantec Internet Security Threat Reports I-XI (January 2002-January 2008)
26. Veldman, F.: Heuristic Anti-Virus Technology. In: International Virus Bulletin Conference, USA, pp. 67–76 (1993)
27. VX Heavens Virus Collection, VX Heavens website, http://vx.netlux.org
28. Walter, S.D.: The partial area under the summary ROC curve. Statistics in Medicine 24(13), 2025–2040 (2005)
29. Witten, I.H., Frank, E.: Data mining: Practical machine learning tools and techniques, 2nd edn. Morgan Kaufmann, USA (2005)

Automatically Adapting a Trained Anomaly Detector to Software Patches

Peng Li[1], Debin Gao[2], and Michael K. Reiter[1]

[1] Department of Computer Science, University of North Carolina, Chapel Hill, NC, USA
[2] School of Information Systems, Singapore Management University, Singapore

Abstract. In order to detect a compromise of a running process based on it deviating from its program's normal system-call behavior, an anomaly detector must first be trained with traces of system calls made by the program when provided clean inputs. When a patch for the monitored program is released, however, the system call behavior of the new version might differ from that of the version it replaces, rendering the anomaly detector too inaccurate for monitoring the new version. In this paper we explore an alternative to collecting traces of the new program version in a clean environment (which may take effort to set up), namely adapting the anomaly detector to accommodate the differences between the old and new program versions. We demonstrate that this adaptation is feasible for such an anomaly detector, given the output of a state-of-the-art binary difference analyzer. Our analysis includes both proofs of properties of the adapted detector, and empirical evaluation of adapted detectors based on four software case studies.

Keywords: Anomaly detection, software patches, system-call monitoring, binary difference analysis.

1 Introduction

One widely studied avenue for detecting the compromise of a process (e.g., by a buffer overflow exploit) is by monitoring its system-call behavior. So-called "white-box" detectors build a model of system-call behavior for the program via static analysis of the source code or binary (e.g., [18,5,11,12,2,13]). "Black-box" (or "gray-box") detectors are trained with system-call traces of the program when processing intended inputs (e.g., [7,6,15,16,9,8]). In either case, deviation of system-call behavior from the model results in an alarm being raised, as this might indicate that the code executing in the process has changed. Both white-box and black/gray-box approaches offer advantages. The hallmark of white-box approaches is the potential for a near-zero or zero false alarm rate [18], if static analysis uncovers every possible system call sequence that the program could possibly emit. Since they are trained on "normal" system-call behavior, black/gray-box approaches can be more sensitive, in that they can reflect nuances of the local environments and usage of the monitored programs [14] and can detect behavioral anomalies that are nevertheless consistent with the control-flow graph of the program. Such anomalies can indicate a compromise (e.g., [3]) and, if ignored, allow more room for mimicry attacks to succeed [19,17].

When a monitored program is patched, an anomaly detector trained on system-call traces may no longer be sufficiently accurate to monitor the updated program. One way

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 142–160, 2009.
© Springer-Verlag Berlin Heidelberg 2009

to address this is to rebuild the model by collecting traces of the updated program. However, these traces must be gathered in a sanitized environment free of attacks that is otherwise as similar as possible — e.g., in terms of the operating system and relevant device configurations and contents, as well as the program usage — to the environment in which the updated program will be run. This problem is compounded if there are multiple such environments.

To avoid the effort of setting up a sanitized environment for collecting system-call traces every time a patch is issued, in this paper we consider an alternative approach to building a model of normal system-call behavior for an updated program. Our approach consists of detecting the differences between the updated program and the previous version, and then directly updating the system-call behavior model to reflect these changes. There are several complexities that arise in doing this, however. First, program patches are often released as wholly new program versions, not isolated patches. Second, in either case, program updates are typically released only in binary format. Both of these make it difficult to detect where the changes occur between versions. Third, while state-of-the-art binary difference analyzers (e.g., [10]) can detect where changes occur, how to modify the system-call model to reflect those changes can require significant further analysis. We emphasize, in particular, that we would like to adapt the model to accommodate these changes while decaying the model's sensitivity to abnormal behavior as little as possible. So, adaptations that increase the model's size (and hence allowed behaviors) more than the changes would warrant should be avoided.

In this paper we provide an algorithm for converting the *execution-graph* anomaly detector [8] on the basis of the output of the BinHunt binary difference analysis tool [10] when applied to a program and its updated version. We show that our algorithm is sound, in the sense that the resulting execution-graph anomaly detector accepts only system-call sequences that are consistent with the control-flow graph of the program. Such soundness was also a requirement of the original execution-graph model [8], and so our algorithm preserves this property of the converted execution graph. In addition, we show through experiments with several patched binaries that our converted execution graphs can be of comparable size to ones generated by training on system-call sequences collected from the updated program, and moreover that the converted execution graphs accept (i.e., do not raise alarms on) those sequences. As such, the converted execution graphs from our algorithms are, based on our experiments, good approximations of the execution graphs that would have been achieved by training. To our knowledge, ours is the first work to automatically update a system-call-based anomaly detection model in response to program patches.

2 Related Work

Systems that employ binary matching techniques to reuse stale "profiles" are most related to our work. Profiles of a program are representatives of how a program is used on a specific machine by a specific user. They usually include program counter information, memory usage, system clock information, etc., and are typically obtained by executing an instrumented version of the program that generates profile information as a side-effect of the program execution. Spike [4] is an optimization system that

collects, manages, and applies profile information to optimize the execution of DEC Alpha executables. When old profiles are used to optimize a new build of a program, Spike simply discards profiles for procedures that have changed, where changes in procedures between two builds of a program are detected by calculating the edit distance between signatures of the corresponding procedures. Spike is not able to re-use profiles of modified procedures.

Wang et al. proposed a binary matching tool, namely BMAT, to propagate profile information from an older, extensively profiled build to a newer build [20]. An optimized version of the newer build is then obtained by applying optimization techniques on the newer build and the propagated profile. The main difference between BMAT and our proposed technique is that we skip the process of propagating the profiles (which roughly correspond to the system-call traces in anomaly detection) and directly propagate the anomaly detection model of the older build to that of the newer build. Our approach is better suited to anomaly detectors that use an automaton-like model because these models are closely related to the control flow of the program (e.g., [8]), and therefore our approach avoids potential inaccuracies introduced in an indirect approach in which system-call traces are derived first.

3 Background and Terminology

To better explain our algorithm for converting the execution-graph anomaly detection model [8], here we provide some background and terminology. We first give our definitions of basic blocks and control flow graphs, which are slightly different from those typical in the literature (c.f., [1]). Next, we outline important concepts in binary difference analysis including common induced subgraphs and relations between two matched basic blocks and two matched functions. We also define important elements in control flow graphs, e.g., call cycles and paths, and finally briefly define an execution graph. The conversion algorithms and their properties presented in Section 4 rely heavily on the definitions and lemmas outlined in this section.

Our definitions below assume that each function is entered only by calling it; jumping into the middle of a function (e.g., using a `goto`) is presumed not to occur. We consider two system calls the same if and only if they invoke the same system-call interface (with potentially different arguments).

Definition 1 [basic block, control-flow subgraph/graph]. A *basic block* is a consecutive sequence of instructions with one entry point. The last instruction in the basic block is the first instruction encountered that is a jump, function call, or function return, or that immediately precedes a jump target.

The *control-flow subgraph* of a function f is a directed graph $\mathsf{cfsg}_f = \langle \mathsf{cfsgV}_f, \mathsf{cfsgE}_f \rangle$. cfsgV_f contains

- a designated f.`enter` node and a designated f.`exit` node; and
- a node per basic block in f. If a basic block ends in a system call or function call, then its node is a *system call node* or *function call node*, respectively. Both types of

nodes are generically referred to as simply *call nodes*. Each node is named by the address immediately following the basic block.[1]

cfsgE_f contains (v, v') if

- $v = f.\mathtt{enter}$ and v' denotes the first basic block executed in the function; or
- $v' = f.\mathtt{exit}$ and v ends with a return instruction; or
- v ends in a jump for which the first instruction of v' is the jump target; or
- the address of the first instruction of v' is the address immediately following (i.e., is the name of) v.

The *control-flow graph* of a program P is a directed graph $\mathsf{cfg}_P = \langle \mathsf{cfgV}_P, \mathsf{cfgE}_P \rangle$ where $\mathsf{cfgV}_P = \bigcup_{f \in P} \mathsf{cfsgV}_f$ and $(v, v') \in \mathsf{cfgE}_P$ iff

- $(v, v') \in \mathsf{cfsgE}_f$ for some $f \in P$; or
- $v' = f.\mathtt{enter}$ for some $f \in P$ and v denotes a basic block ending in a call to f; or
- $v = f.\mathtt{exit}$ for some $f \in P$ and v' denotes a basic block ending in a call to f. □

We next define common induced subgraphs, which are used in binary difference analysis of two programs [10].

Definition 2 [common induced subgraph, $\sim$, $\approx$]. Given $\mathsf{cfsg}_f = \langle \mathsf{cfsgV}_f, \mathsf{cfsgE}_f \rangle$, an *induced subgraph* of cfsg_f is a graph $\mathsf{isg}_f = \langle \mathsf{isgV}_f, \mathsf{isgE}_f \rangle$ where $\mathsf{isgV}_f \subseteq \mathsf{cfsgV}_f$ and $\mathsf{isgE}_f = \mathsf{cfsgE}_f \cap (\mathsf{isgV}_f \times \mathsf{isgV}_f)$. Given two functions f and g, a *common induced subgraph* is a pair $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$ of induced subgraphs of cfsg_f and cfsg_g, respectively, that are isomorphic. We use $\sim$ to denote the node isomorphism; i.e., if $v \in \mathsf{isgV}_f$ maps to $w \in \mathsf{isgV}_g$ in the isomorphism, then we write $v \sim w$ and say that v "matches" w. Similarly, if $v \sim w$, $v' \sim w'$, and $(v, v') \in \mathsf{isgE}_f$ (and so $(w, w') \in \mathsf{isgE}_g$), then we write $(v, v') \sim (w, w')$ and say that edge (v, v') "matches" (w, w').

The algorithm presented in this paper takes as input an injective partial function $\pi : \{f : f \in P\} \to \{g : g \in Q\}$ for two programs P and Q, and induced subgraphs $\{\langle \mathsf{isg}_f, \mathsf{isg}_{\pi(f)} \rangle : \pi(f) \neq \perp\}$. We naturally extend the "matching" relation to functions by writing $f \sim \pi(f)$ if $\pi(f) \neq \perp$, and say that f "matches" $\pi(f)$. Two matched functions f and g are *similar*, denoted $f \approx g$, iff $\mathsf{isg}_f = \mathsf{cfsg}_f$ and $\mathsf{isg}_g = \mathsf{cfsg}_g$. □

Control-flow subgraphs and graphs, and common induced subgraphs for two programs, can be extracted using static analysis of binaries [10]. When necessary, we will appeal to static analysis in the present work, assuming that static analysis is able to disassemble the binary successfully to locate the instructions in each function, and to build cfsg_f for all functions f and cfg_P for the program P.

A tool that provides the common induced subgraphs required by our algorithm is BinHunt [10]. When two nodes are found to match each other by BinHunt, they are functionally similar. For example, if $v \in \mathsf{isgV}_f$, $w \in \mathsf{isgV}_{\pi(f)}$, and $v \sim w$, then either both v and w are call nodes, or neither is; we utilize this property in our algorithm. However, BinHunt compares two nodes by analyzing the instructions *within* each node only, and so the meaning of *match* does not extend to functions called by the nodes. For example, two nodes, each of which contains a single `call` instruction, may match

[1] For a function call node, this name is the return address for the call it makes.

to each other even if they call very different functions. In order to extend the meaning of *match* to functions called by the nodes, we introduce a new relation between two functions (and subsequently two nodes), called *extended similarity*.

Definition 3 [$\overset{\text{ext}}{\approx}$]. Two matched functions f and g are *extended-dissimilar*, denoted $f \overset{\text{ext}}{\not\approx} g$, iff

– (Base cases)
  • $f \not\approx g$; or
  • for two system call nodes $v \in \mathsf{cfsg}_f$ and $w \in \mathsf{cfsg}_g$ such that $v \sim w$, v and w call different system calls; or
  • for two function call nodes $v \in \mathsf{cfsg}_f$ and $w \in \mathsf{cfsg}_g$ such that $v \sim w$, if v calls f' and w calls g', then $f' \not\approx g'$.

– (Induction) For two function call nodes $v \in \mathsf{cfsg}_f$ and $w \in \mathsf{cfsg}_g$ such that $v \sim w$, if v calls f' and w calls g', then $f' \overset{\text{ext}}{\not\approx} g'$.

If two matched functions f and g are not extended-dissimilar, then they are *extended-similar*, denoted $f \overset{\text{ext}}{\approx} g$. Two matched nodes v and w are *extended-similar*, denoted $v \overset{\text{ext}}{\approx} w$, if (i) neither v nor w is a call node; or (ii) v and w make the same system call; or (iii) v and w call f and g, respectively, and $f \overset{\text{ext}}{\approx} g$. □

Two extended-similar nodes exhibit a useful property that will be stated in Lemma 1. To state this property, we first define call cycles.

Definition 4 [Call cycle]. A sequence of nodes $\langle v_1, \ldots, v_l \rangle$ in cfg_P is a *call cycle from* v iff for some function $f \in P$, $v = v_1 = v_l$ is a function call node calling to f, $v_2 = f.\mathtt{enter}$, $v_{l-1} = f.\mathtt{exit}$, and

– (Base case) For each $i \in (1, l-1)$, $v_i \in \mathsf{cfsgV}_f$ and $(v_i, v_{i+1}) \in \mathsf{cfsgE}_f$.
– (Induction) For some $k, k' \in (1, l-1)$, $k < k'$,
  • for each $i \in (1, k] \cup [k', l)$, $v_i \in \mathsf{cfsgV}_f$; and
  • for each $i \in (1, k) \cup [k', l-1)$, $(v_i, v_{i+1}) \in \mathsf{cfsgE}_f$; and
  • $\langle v_k, \ldots, v_{k'} \rangle$ is a call cycle from $v_k = v_{k'}$. □

Lemma 1. *If v and w are call nodes in P and Q, respectively, and $v \overset{\text{ext}}{\approx} w$, then for every call cycle from v that results in a (possibly empty) sequence of system calls, there is a call cycle from w that results in the same sequence of system calls.*

Lemma 1, which is proved in Appendix B, shows a useful property about extended-similar nodes, and is used in our proofs of properties of the converted execution graph. As we will see, some edges can be copied from the execution graph of the old binary P to the execution graph of the new binary Q on the basis of nodes in cfg_P being extended-similar to nodes in cfg_Q, since those nodes exhibit similar system-call behavior. Next, we define paths to help refer to sequences of nodes in a control flow graph.

Definition 5 [Path, full, pruned, silent, audible]. A *path* $p = \langle v_1, \ldots, v_n \rangle$ is a sequence of nodes where

– for all $i \in [1, n]$, $v_i \in \mathsf{cfgV}_P$; and
– for all $i \in [1, n)$, $(v_i, v_{i+1}) \in \mathsf{cfgE}_P$.

We use $|p|$ to denote the length of p which is n.

p is *pruned* if no $v \in \{v_2, \ldots, v_n\}$ is a function `enter` node, and if no $v \in \{v_1, \ldots, v_{n-1}\}$ is a function `exit` node. p is *full* if for every function call node $v \notin \{v_1, v_n\}$ on p, v is either followed by a function `enter` node or preceded by a function `exit` node (but not both).

p is called *silent* if for all $i \in (1, n)$, v_i is not a system call node. Otherwise, it is called *audible*. □

Next, we define an execution graph [8], which is a model for system-call-based anomaly detection. We begin with two technical definitions, however, that simplify the description of an execution graph.

Definition 6 [Entry call node, exit call node]. A node $v \in \mathsf{cfsg}_f$ is an *entry call node* of f if v is a call node and there exists a full silent path $p = \langle f.\mathtt{enter}, \ldots, v\rangle$. A node $v \in \mathsf{cfsg}_f$ is an *exit call node* of f if v is a call node and there exists a full silent path $p = \langle v, \ldots, f.\mathtt{exit}\rangle$. □

Definition 7 [support ($\rightsquigarrow$), strong support ($\overset{\mathsf{s}}{\rightsquigarrow}$)]. A (full or pruned) path $p = \langle v, \ldots, v'\rangle$ *supports* an edge (v, v'), denoted $p \rightsquigarrow (v, v')$, if p is silent. p *strongly supports* (v, v'), denoted $p \overset{\mathsf{s}}{\rightsquigarrow} (v, v')$, if $p \rightsquigarrow (v, v')$ and if each of v and v' is a system call node or a function call node from which there is at least one audible call cycle. □

Definition 8 [Execution subgraph/graph]. An *execution subgraph* of a function f is a directed graph $\mathsf{esg}_f = \langle \mathsf{esgV}_f, \mathsf{esgE}_f\rangle$ where $\mathsf{esgV}_f \subseteq \mathsf{cfsgV}_f$ consists only of call nodes. If $(v, v') \in \mathsf{esgE}_f$ then there is a full path $p = \langle v, \ldots, v'\rangle$ such that $p \overset{\mathsf{s}}{\rightsquigarrow} (v, v')$.

An *execution graph* of a program P is a directed graph $\mathsf{eg}_P = \langle \mathsf{egV}_P$, egEcl_P, egEcr_P, $\mathsf{egErt}_P\rangle$ where egEcl_P, egEcr_P, and egErt_P are sets of *call edges*, *cross edges* and *return edges*, respectively. $\mathsf{egV}_P = \bigcup_{f \in P} \mathsf{esgV}_f$ and $\mathsf{egEcr}_P = \bigcup_{f \in P} \mathsf{esgE}_f$. If $(v, v') \in \mathsf{egEcl}_P$, then v is a function call node ending in a call to the function f containing v', and v' is a entry call node. If $(v', v) \in \mathsf{egErt}_P$, then v is a function call node ending in a call to the function f containing v', and v' is an exit call node. □

An execution graph eg_P is built by subjecting P to a set of legitimate inputs in a protected environment, and recording the system calls that are emitted and the return addresses on the function call stack when each system call is made. This data enables the construction of an execution graph. Then, to monitor a process ostensibly running P in the wild, the return addresses on the stack are extracted from the process when each system call is made. The monitor determines whether the sequence of system call (and the return addresses when those calls are made) are consistent with traversal of a path in eg_P. Any such sequence is said to be in the *language accepted by the execution graph*. Analogous monitoring could be performed using cfg_P, instead, and so we can similarly define a *language accepted by the control flow graph*. An execution graph eg_P is built so that any sequence in its language is also in the language accepted by cfg_P [8].

4 The Conversion Algorithm

Suppose that we have an execution graph eg_P for a program P, and that a patch to P is released, yielding a new program Q. In this section, we show our conversion algorithm to obtain eg_Q. In addition to utilizing eg_P, our algorithm utilizes the output of a binary difference analysis tool (e.g., [10]), specifically a partial injective function π and pairs $\langle \mathsf{isg}_f, \mathsf{isg}_{\pi(f)} \rangle$ of isomorphic induced subgraphs. Our algorithm also selectively uses static analysis on Q. Unless stated otherwise, below we use f, v and p to denote a function, node and path, respectively, in cfg_P, and we use g, w, and q to denote a function, node and path, respectively, in cfg_Q. In addition, we abuse notation in using "$\in$" to denote a path being in a graph (e.g., "$p \in \mathsf{cfg}_P$"), in addition to its normal use for set membership.

Recall that we have two important requirements in designing the conversion algorithm. A first is that eg_Q preserves the soundness property of the original execution-graph model, namely that it accepts only system-call sequences that are consistent with cfg_Q. A second requirement is that it decays the model's sensitivity to abnormal behavior as little as possible, and therefore preserves the advantage of black-box and gray-box models in that eg_Q should not accept system-call behavior that would not have been observed were it built by training, even though this behavior may be accepted by cfg_Q.

We satisfy the above two requirements by

- creating counterparts of as many nodes and edges in eg_P as possible in eg_Q;
- adding new nodes and edges to eg_Q to accommodate changes between P and Q; and
- performing the above two tasks in such a way that a minimal set of system-call behaviors is accepted by eg_Q.

More specifically, we first copy matched nodes and edges in esg_f to esg_g to the extent possible for all matched function pairs $f \sim g$ (Section 4.1). Next, we handle nodes in cfsg_g that are not matched and create corresponding cross edges (Section 4.2). In the last two steps, we further process the function call nodes to account for the functions they call (Section 4.3) and connect execution subgraphs together to obtain the execution graph eg_Q (Section 4.4).

4.1 Copying Nodes and Edges When $f \sim g$

The first step, called $\mathsf{copy}()$, in our conversion algorithm is to copy matched portions in esg_f to esg_g, if $f \sim g$. This is an important step as it is able to obtain a large portion of eg_Q, assuming that there is little difference between P and Q, and that the binary difference analysis that precedes our conversion produces common induced subgraphs $\langle \mathsf{isg}_f, \mathsf{isg}_{\pi(f)} \rangle$ that are fairly complete for most $f \in P$. Intuitively, for two matched functions f and g, we simply need to copy all nodes and edges in esg_f that are matched and update the names of the nodes (which denote return addresses). However, when a cross edge is copied to esg_g, we need to make sure that there is a full path in cfg_Q that can result in the newly added cross edge (i.e., to make sure that it is supported by a full path).

There are two caveats to which we need to pay attention. The first is that a cross edge in esg_f supported by a pruned path containing edges in $\mathsf{cfsgE}_f \setminus \mathsf{isgE}_f$ should not be

copied to esg_g, because something has changed on this pruned path and may render the cross edge not supported in cfg_Q. To improve efficiency, here we restrict our analysis within f and g only and require that all pruned paths (instead of full paths) supporting the cross edge to be copied be included in isg_f and isg_g.

For the example in Figure 1, a cross edge (v, v') is supported by the pruned path $\langle v, v_2, v_3, v' \rangle$ in cfsg_f (which is also a full path). However, there is no pruned path in isg_g that supports the corresponding cross edge in esg_g (so no full path in cfg_Q will support it). The only pruned path $\langle w, w_2, w_4, w' \rangle$ in isg_g does not support this cross edge since this pruned path would unavoidably induce a system call. Thus, the cross edge (v, v') cannot be copied to esg_g.

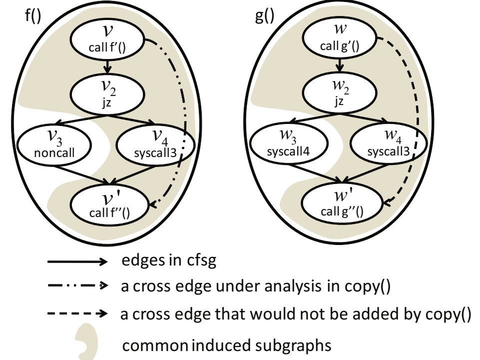


Fig. 1. Cross edge that is not copied

A second caveat is related to the notion of extended similarity that we introduced in Section 3. Assume $v \sim w$, $v' \sim w'$, and $v'' \sim w''$ (see Figure 2); also assume that $\langle v, v'', v' \rangle \rightsquigarrow (v, v')$. To copy (v, v') to esg_g, we need $\langle w, w'', w' \rangle \rightsquigarrow (w, w')$ and therefore $v'' \overset{\text{ext}}{\approx} w''$ so that any call cycle from v'' can be "replicated" by a call cycle from w'', yielding the same system-call behavior (c.f., Lemma 1).

In summary, when a cross edge (w, w') is created in esg_g in this step, all the nodes on the pruned paths supporting this edge are matched, and the nodes along each pruned path not only match but are extended-similar if they are call nodes. We are very strict when copying a cross edge because we do not know which one of the many supporting pruned paths was taken during training of eg_P. In order to avoid possible mistakes in copying a cross edge to esg_g that 1) is not supported by a full path in cfg_Q; or 2) would not have been created had training been done on Q, we have to require that all nodes on all supporting pruned paths be matched and extended-similar. In Figure 3, three cross edges are copied since all the pruned paths that support them are in the common induced subgraph and call nodes are extended-similar.

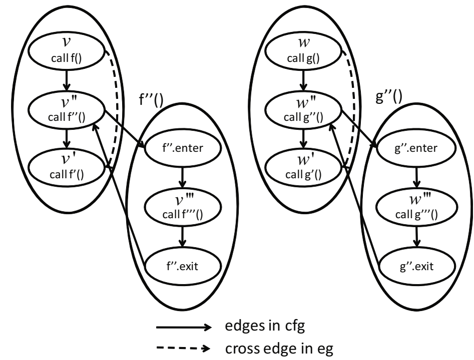


Fig. 2. Extended similarity in copy()

Algorithm 1 copy(), in Appendix A, performs the operations in this step to copy nodes and cross edges. The following holds for the cross edges it copies to esg_g^{cp}.

Lemma 2. *Every cross edge added by* copy() *is strongly supported by a full path in* cfg_Q.

Please refer to Appendix B for an outline of the proof.

copy() creates nodes and cross edges by copying them from esg_f. The next step (Section 4.2) shows how we create more nodes and edges for esg_g by statically analyzing the unmatched portion of g.

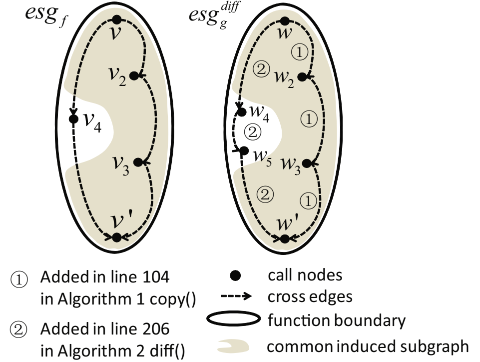


Fig. 3. Converting an execution subgraph

4.2 The Unmatched Portion of g

Assuming that f and $g = \pi(f)$ differ by only a small portion, copy() would have created most of the nodes and cross edges for esg_g. In this step, we analyze the unmatched portion of g to make esg_g more complete. This step is necessary because esg_f does not contain information about the difference between f and g. Intuitively, esg_f and $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$ do not provide enough information for dealing with the unmatched portion of g, and we need to get help from static analysis.

We identify each pruned path in cfsg_g that passes through the unmatched portion of g and then build cross edges between consecutive call nodes on this pruned path until this path is connected to the nodes we created in Algorithm 1 copy(). Three cross edges in Figure 3 are created in this way due to the unmatched nodes w_4 and w_5.

This algorithm, diff(), is detailed in Appendix A. diff() results in the following property for the cross edges it adds to $\mathsf{esg}_g^{\mathsf{diff}}$; Appendix B gives an outline of the proof.

Lemma 3. *Every cross edge added by* diff() *is supported by a full path in* cfg_Q.

If there is a cross edge in esg_f that was not copied by copy() to esg_g, this occurred because a supporting pruned path for this edge was changed (containing unmatched nodes or nodes that are matched but not extended-similar) in g. Whether this pruned path was traversed when P emitted the system-call sequences on which eg_P was trained is, however, unknown. One approach to decide whether to copy the cross edge to esg_g is to exhaustively search (e.g., in diff()) for a full path in cfg_Q that supports it. That is, *any* such path is taken as justification for the cross edge; this approach, therefore, potentially decreases the sensitivity of the model (and also, potentially, false alarms). Another possibility, which reflects the version of the algorithm in Appendix A, is to copy the cross edge only if there is a full supporting path that involves the changed (unmatched) part of g. (This process is encompassed by refine() in Appendix A, described below in Section 4.3.) In addition to this approach sufficing in our evaluation in Section 5, it better preserves the sensitivity of the model.

4.3 Refining esg_g Based on Called Functions

Many function call nodes have been created in esg_g by copy() and diff(). Except those extended-similar to their counterparts in cfsgV_f, many of these nodes are created

without considering the system-call behavior of the called functions. This is the reason why Lemma 3 claims only that the cross edges created are *supported* but not *strongly supported*. In this step, called $\mathsf{refine}()$, we analyze the system-call behavior of the corresponding called functions and extend the notion of support to strong support for cross edges created so far in $\mathsf{copy}()$ and $\mathsf{diff}()$.

An obvious case in which function call nodes need more processing is when the execution subgraph of the called function has not been created. This happens when the called function g' does not have a match with any function in P. In this case, $\mathsf{esg}_{g'}$ can be obtained by statically analyzing the function itself. For simplicity in this presentation, we reuse $\mathsf{diff}()$ to denote this process in Appendix A, with empty sets for the first three arguments, i.e., $\mathsf{diff}(\emptyset, \langle\emptyset, \emptyset\rangle, \mathsf{cfsg}_{g'})$.

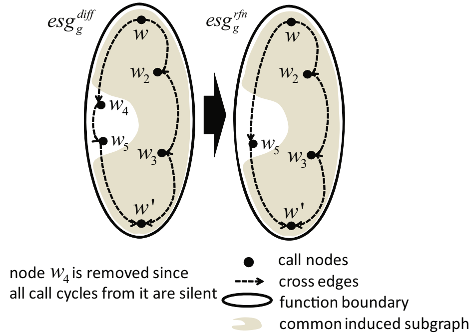


Fig. 4. Function call node removed and cross edges modified

Another scenario in which the function call nodes need more processing is when the called function does not make a system call. Recall that a call node w is created in $\mathsf{copy}()$ and $\mathsf{diff}()$ but we might not have analyzed the called function g' at that time and simply assumed that system calls are made in g' (and therefore these cross edges are *supported* instead of being *strongly supported*). If g' may not make a system call, then we need to either delete w (in the case where g' never makes a system call, shown in Figure 4 where all call cycles from w_4 are silent) or add cross edges from predecessor call nodes of w to successor call nodes of w (in the case where g' may or may not make a system call).

Lemma 4. *After* $\mathsf{refine}()$*, every cross edge in* esg_g *is strongly supported by a full path in* cfg_Q.

Please refer to Appendix B for the proof of Lemma 4.

4.4 Connecting Execution Subgraphs

At this stage, we create call and return edges to connect all esg_g to form eg_Q. Some of these call edges are created by "copying" the edges from the eg_P, e.g., when the corresponding call node is created in $\mathsf{copy}()$ and is extended-similar to its counterpart in eg_P (case 1 in Figure 5, where $f' \overset{\mathsf{ext}}{\approx} g'$). If a call node w has a match v but is not extended-similar to it, we create an edge (w, w') only for each entry call node w' in the function called by w that matches an entry call node v' for which $(v, v') \in \mathsf{egEcl}_P$ (case 2 in Figure 5, where $f'' \overset{\mathsf{ext}}{\not\approx} g''$), or to all entry call nodes in the called function if there is no such v'. For other call nodes, the call and return edges cannot be created via copying, and we add call edges between this call node and all the entry call nodes of the called function (case 3 in Figure 5). We create return edges in a similar way.

Appendix A briefly gives an implementation of connect(), and please refer to Appendix B for an outline of the proof of Lemma 5.

Lemma 5. *Every call or return edge added by* connect() *is strongly supported by a full path in* cfg_Q.

Therefore, after running our conversion algorithm, we have a converted execution graph of the new program eg_Q with all the nodes being system call nodes or function call nodes with at least one audible call cycle from each, and all the edges being strongly supported by cfg_Q. Finally, we can state the soundness of our conversion algorithm:

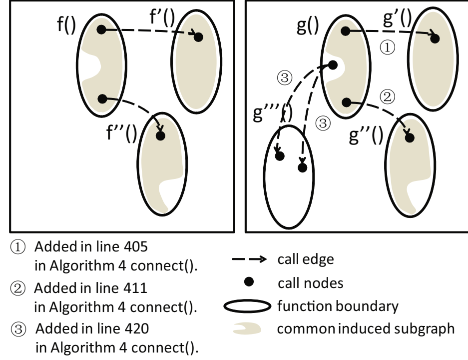


Fig. 5. Using call and return edges to connect execution subgraphs

Lemma 6. *The language accepted by* eg_Q *is a subset of the language accepted by* cfg_Q.

This result is trivial given Lemmas 2–5, and consists primarily in arguing that any path q traversed in eg_Q can be "mimicked" by traversing a full path in cfg_Q that travels from each node of q to the next, say from w to w', by following the full path in cfg_Q that strongly supports (w, w').

5 Evaluation

In this section, we evaluate the performance of our conversion procedure. Our conversion program takes in the execution graph of the old binary eg_P, the control flow graph for both binaries cfg_P and cfg_Q, and the output of the binary difference analyzer BinHunt, and outputs the converted execution graph eg_Q of the new binary. We implemented Algorithms 1-4 with approximately 3000 lines of Ocaml code.

We evaluated execution graphs obtained by our conversion algorithm by comparing them to alternatives. Specifically, for each case study, we compared the converted execution graph for the patched program Q with (i) an execution graph for Q obtained by training and (ii) the control flow graph of Q. We performed four case studies.

tar. Version 1.14 of tar (P) has an input validation error. Version 1.14-2.3 (Q) differs from P by changing a **do** {} **while**( ) loop into a **while**( ) **do** {} loop (see `http://www.securityfocus.com/bid/25417/info`). This change is identified by BinHunt, but it involves only a function call that does not make any system calls. As such, the system-call behavior of the two programs remains unchanged, and so does the execution graph obtained by our conversion algorithm. (diff() adds a new node and the corresponding cross edges for the function call involved in the change, which are subsequently deleted in refine() because all call cycles from it are silent.)

ncompress. In version 4.2.4 of ncompress (P), a missing boundary check allows a specially crafted data stream to underflow a buffer with attacker's data. A check was

added in version 4.2.4-15 (Q) to fix this problem (see `http://www.debian.org/security/2006/dsa-1149`). The check introduces a new branch in the program in which an error message is printed when the check fails, causing a new system call to be invoked. With the same benign inputs for training, the execution graphs for both programs are the same. Our conversion algorithm, however, tries to include this new branch by performing limited static analysis, and consequently expands the execution graph by 3 nodes and 23 edges.

ProFTPD. ProFTPD version 1.3.0 (P) interprets long commands from an FTP client as multiple commands, which allows remote attackers to conduct cross-site request forgery (CSRF) attacks and execute arbitrary FTP commands via a long `ftp://` URI that leverages an existing session from the FTP client implementation in a web browser. For the stable distribution (etch) this problem has been fixed in version 1.3.0-19etch2 (Q) by adding input validation checks (see `http://www.debian.org/security/2008/dsa-1689`). Eight additional function calls are introduced in the patched part, most to a logging function for which the execution subgraph can be copied from the old model. The converted execution graph for the patched version thus only slightly increases the execution graph size.

unzip. When processing specially crafted ZIP archives, unzip version 5.52 (P) may pass invalid pointers to a C library's `free()` routine, potentially leading to arbitrary code execution (CVE-2008-0888). A patch (version 5.52-1 (Q)) was issued with changes in four functions (see `http://www.debian.org/security/2008/dsa-1522`). Some of the changes involve calling to a new function for which there is no corresponding execution subgraph for the old version. All four changes resulted in static analysis in our conversion algorithm, leading to execution subgraphs constructed mostly or entirely by static analysis. This increased the number of nodes and edges in the resulting execution graph eg_Q more significantly compared to the first three cases.

Experimental results are shown in Table 1 and Table 2. In Table 1, we show the number of nodes and edges in eg_Q that have their counterparts in eg_P and those that do not. More precisely, if $w \in \mathsf{egV}_Q$ and there is some $v \in \mathsf{egV}_P$ such that $v \sim w$, then w is accounted for in the "borrowed" column in Table 1. Similarly, if $(w, w') \in \mathsf{egEcl}_Q \cup \mathsf{egErt}_Q \cup \mathsf{egEcr}_Q$ and there is some $(v, v') \in \mathsf{egEcl}_P \cup \mathsf{egErt}_P \cup \mathsf{egEcr}_P$ such that $(v, v') \sim (w, w')$, then (w, w') is accounted for in the "borrowed" column. Nodes and edges in eg_Q not meeting these conditions are accounted for in the "not borrowed" columns. As this table shows, increased use of static analysis (e.g., in the case of unzip) tends to inflate the execution graph.

Table 1. Evaluation: nodes and edges in eg_Q

	borrowed from eg_P		not borrowed from eg_P	
	# of nodes	# of edges	# of nodes	# of edges
tar	478	1430	0	0
ncompress	151	489	3	23
ProFTPD	775	1850	6	28
unzip	374	1004	50	195

Table 2 compares eg_Q obtained by conversion with one obtained by training. As we can see, eg_Q obtained by training is only marginally smaller than the one obtained by conversion for the first three cases. They differ slightly more in size in the unzip

Table 2. Statistics for four case studies. Numbers of nodes for eg_P and eg_Q are highlighted as representatives for size comparison.

model	Old binary P				New binary Q						
	eg_P (trained)		cfg_P		eg_Q (converted)			eg_Q (trained)		cfg_Q	
	nodes	edges	nodes	edges	nodes	edges	time (s)	nodes	edges	nodes	edges
tar	**478**	1430	2633	7607	**478**	1430	14.5	**478**	1430	2633	7607
ncompress	**151**	489	577	1318	**154**	512	13.1	**151**	489	578	1322
ProFTPD	**775**	1850	3343	9160	**781**	1878	17.4	**776**	1853	3351	9193
unzip	**374**	1004	491	1464	**424**	1199	41.6	**377**	1017	495	1490

case, due to the more extensive use of static analysis. When the eg_Q as obtained by conversion is substantially larger than eg_P, as in the unzip case, this is an indication that rebuilding eg_Q by training might be prudent.

Both converted eg_Q and trained eg_Q are smaller than cfg_Q, which, in our experiments, includes cfsg_g for each g reachable from the first function executed in the binary, including library functions. The numbers presented for cfg_Q do *not* include non-call nodes, function call nodes that do not give rise to audible call cycles, `enter` nodes, or `exit` nodes, to enable a fair comparison with eg_Q (since eg_Q does not contain these nodes). Since eg_Q, when trained, is a function of the training inputs, the gap between the sizes of cfg_Q and eg_Q would presumably narrow somewhat by training on a wider variety of inputs (though we did endeavor to train thoroughly, see Appendix C). Absolute sizes aside, however, Table 2 suggests that our conversion algorithm often retains the precision offered by the execution graph from which it builds, no matter how well (or poorly) trained.

An important observation about our converted execution graphs in these case studies is that the language each accepts includes all system-call sequences output by Q when provided the training inputs. We cannot prove that this will always hold with our conversion algorithm, due to limitations on the accuracy of the binary difference analysis tool from which we build [10]. Nevertheless, this empirically provides evidence that this property should often hold in practice.

The conversion time shown in Table 2 for each eg_Q (converted) is in seconds on a 2.8 GHz CPU platform with 1GB memory, and includes only our algorithm time, excluding binary difference analysis and the construction of cfg_Q. (Binary difference analysis with BinHunt overwhelmingly dominated the total conversion time.) As shown in Table 2, as the changes between P and Q increase in size, more time is spent on analyzing cfg_Q and building eg_Q statically. In the cases of ncompress and unzip, the static analysis needs to be applied to the libraries as well.

6 Conclusion

We have presented an algorithm by which an *execution graph*, which is a gray-box system-call-based anomaly detector that uses a model trained from observed system-call behaviors, can be converted from the program for which it was originally trained to a patched version of that program. By using this algorithm, administrators can be

spared from setting up a protected and identically configured environment for collecting traces from the patched program. Our algorithm retains desirable properties of execution graphs, including that the system-call sequences accepted by the execution graph are also consistent with the control-flow graph of the program, and that the sequences accepted tend to capture "normal" behavior as defined by the training sequences. We have demonstrated the effectiveness of our algorithm with four case studies.

As our paper is the first to study adapting anomaly detectors to patches, we believe it introduces an important direction of new research. There are numerous system-call-based anomaly detectors in the literature. Our initial studies suggest that many other such detectors pose challenges to conversion beyond those we have addressed here.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Basu, S., Uppuluri, P.: Proxi-annotated control flow graphs: Deterministic context-sensitive monitoring for intrusion detection, pp. 353–362. Springer, Heidelberg (2004)
3. Buchanan, E., Roemer, R., Schacham, H., Savage, S.: When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (October 2008)
4. Cohn, R.S., Goodwin, D.W., Lowney, P.G.: Optimizing Alpha executables on Windows NT with Spike. Digital Tech. J. 9, 3–20 (1998)
5. Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., Miller, B.: Formalizing sensitivity in static analysis for intrusion detection. In: Proceedings of the 2004 IEEE Symposium on Security and Privacy (May 2004)
6. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy, May 2003, pp. 62–75 (2003)
7. Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T.: A sense of self for Unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, May 1996, pp. 120–128 (1996)
8. Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graph for anomaly detection. In: Proceedings of the 11th ACM Conference on Computer & Communication Security (CCS 2004) (2004)
9. Gao, D., Reiter, M.K., Song, D.: On gray-box program tracking for anomaly detection. In: Proceedings of the 13th USENIX Security Symposium (2004)
10. Gao, D., Reiter, M.K., Song, D.: BinHunt: Automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008)
11. Giffin, J., Jha, S., Miller, B.: Detecting manipulated remote call streams. In: Proceedings of the 11th USENIX Security Symposium (August 2002)
12. Giffin, J., Jha, S., Miller, B.: Efficient context-sensitive intrusion detection. In: Proceedings of the ISOC Symposium on Network and Distributed System Security (February 2004)
13. Gopalakkrishna, R., Spafford, E.H., Vitek, J.: Efficient intrusion detection using automaton inlining. In: Proceedings of the 2005 Symposium on Security and Privacy, pp. 18–31 (2005)
14. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security, 151–180 (1998)

15. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, May 2001, pp. 144–155 (2001)
16. Tan, K., Maxion, R.: "Why 6?"– Defining the operational limits of stide, an anomaly-based intrusion detector. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002, pp. 188–201 (2002)
17. Tan, K., McHugh, J., Killourhy, K.: Hiding intrusions: From the abnormal to the normal and beyond. In: Petitcolas, F.A.P. (ed.) IH 2002. LNCS, vol. 2578, pp. 1–17. Springer, Heidelberg (2003)
18. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy (May 2001)
19. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security (2002)
20. Wang, Z., Piece, K., Mcfarling, S.: BMAT – a binary matching tool for stale profile propagation. The Journal of Instruction-Level Parallelism 2(2000) (2000)

A Algorithms

The notation used in the following algorithms follow the convention we stated at the beginning of Section 4: we use f, v and p to denote a function, node and path, respectively, in cfg_P, and we use g, w, and q to denote a function, node and path, respectively, in cfg_Q. We also continue to use $\in$ to denote not only set membership, but a path being in a graph, as well.

Algorithm 1 copy() picks cross edges from the old function execution subgraph, when we have matches for the two ends of a cross edge and when there is no change that would potentially affect this edge. We copy the edge into the new function execution subgraph (line 104).

Algorithm 1. copy()

Input: esg_f, $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$, cfsg_f, cfsg_g
100: **for all** $(v, v') \in \mathsf{esgE}_f$ **do**
101: **if** $\exists w, w' : v \sim w$ and $v' \sim w'$ **then**
102: $\mathsf{esgV}_g^{\mathsf{cp}} \leftarrow \mathsf{esgV}_g^{\mathsf{cp}} \cup \{w, w'\}$
103: **if** $\forall p \in \mathsf{cfsg}_f, p \overset{\mathsf{s}}{\rightsquigarrow} (v, v')\ \exists q \in \mathsf{isg}_g : \forall v'' \in p\ \exists w'' \in q : v'' \overset{\mathsf{ext}}{\approx} w''$ **then**
104: $\mathsf{esgE}_g^{\mathsf{cp}} \leftarrow \mathsf{esgE}_g^{\mathsf{cp}} \cup \{(w, w')\}$
Output: $\mathsf{esg}_g^{\mathsf{cp}}$

In this implementation of Algorithm 1, we examine all pruned paths that strongly support the cross edge to be copied to esg_g (line 103). When the two functions f and g are similar, it is more efficient to examine the differences between f and g to discover the cross edges that should not be copied. When the differences between f and g are small, this equivalent algorithm is more efficient, in our experience.

Algorithm 2 diff() modifies $\mathsf{esg}_g^{\mathsf{cp}}$ created in copy(). It analyzes each pruned path that passes through the unmatched portion of g, and tries to create a part of execution graph along each such pruned path and connect it to the rest of the execution subgraph.

Algorithm 2. diff()

Input: $\mathsf{esg}_g^{\mathsf{cp}}$, $\langle \mathsf{isg}_f, \mathsf{isg}_g \rangle$, cfsg_g
200: $\mathsf{esg}_g^{\mathsf{diff}} \leftarrow \mathsf{esg}_g^{\mathsf{cp}}$
201: $U \leftarrow \{w \mid w \in \mathsf{cfsgV}_g \ \wedge (w \notin \mathsf{isgV}_g \vee (\exists v : v \sim w \wedge v \overset{\mathsf{ext}}{\not\approx} w))\}$
202: $U' \leftarrow \{w \mid w \in \mathsf{esgV}_g^{\mathsf{cp}} \vee (w \in U \wedge w$ is a call node$)\}$
203: **for all** $w \in U$ **do**
204: **for all** $q = \langle w_1, \ldots, w_{|q|} \rangle \in \mathsf{cfsg}_g : w \in q \wedge$ $(\forall i \in (1, |q|) : w_i \neq w \Rightarrow w_i \notin U') \ \wedge \{w_1, w_{|q|}\} \subseteq U'$ **do**
205: $\mathsf{esgV}_g^{\mathsf{diff}} \leftarrow \mathsf{esgV}_g^{\mathsf{diff}} \ \cup \{w_i \mid i \in [1, |q|] \ \wedge w_i$ is a call node$\}$
206: $\mathsf{esgE}_g^{\mathsf{diff}} \leftarrow \mathsf{esgE}_g^{\mathsf{diff}} \ \cup \{(w_i, w_j) \mid \ i, j \in [1, |q|] \ \wedge w_i, w_j$ are call nodes $\wedge\, i < j \,\wedge$ $\forall k \in (i, j) : w_k$ is not a call node$\}$
Output: $\mathsf{esg}_g^{\mathsf{diff}}$

Algorithm 3. refine()

Input: $\{\mathsf{esg}_g^{\mathsf{cp}}\}_g, H = \{\mathsf{esg}_g^{\mathsf{diff}}\}_g, \mathsf{cfg}_Q$
300: **while** $H \neq \emptyset$ **do**
301: pick one $\mathsf{esg}_g^{\mathsf{diff}}$ in H
302: $\mathsf{esg}_g^{\mathsf{rfn}} \leftarrow \mathsf{esg}_g^{\mathsf{diff}}$
303: **for all** $w \in \mathsf{esgV}_g^{\mathsf{rfn}}$: w is a function call node $\wedge$ $w \notin \left\{ w' \mid w' \in \mathsf{esgV}_g^{\mathsf{cp}} \wedge \exists v' : v' \overset{\mathsf{ext}}{\approx} w' \right\}$ **do**
304: let g' be the function called by w
305: **if** no call cycle from w is audible **then**
306: $\mathsf{esgV}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgV}_g^{\mathsf{rfn}} \setminus \{w\}$
307: **for all** w', w'' : $(w', w) \in \mathsf{esgE}_g^{\mathsf{rfn}} \ \wedge (w, w'') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**
308: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \cup \{(w', w'')\}$
309: **for all** w' : $(w, w') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**
310: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \setminus \{(w, w')\}$
311: **for all** w' : $(w', w) \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**
312: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \setminus \{(w', w)\}$
313: **else if** all call cycles from w are audible **then**
314: **if** $\mathsf{esg}_{g'}^{\mathsf{diff}} \notin H$ **then**
315: $H \leftarrow H \cup \{\mathsf{diff}(\emptyset, \langle \emptyset, \emptyset \rangle, \mathsf{cfsg}_{g'})\}$
316: **else**
317: **for all** w', w'' : $(w', w) \in \mathsf{esgE}_g^{\mathsf{rfn}} \ \wedge (w, w'') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ **do**
318: $\mathsf{esgE}_g^{\mathsf{rfn}} \leftarrow \mathsf{esgE}_g^{\mathsf{rfn}} \cup \{(w', w'')\}$
319: **if** $\mathsf{esg}_{g'}^{\mathsf{diff}} \notin H$ **then**
320: $H \leftarrow H \cup \{\mathsf{diff}(\emptyset, \langle \emptyset, \emptyset \rangle, \mathsf{cfsg}_{g'})\}$
321: $H \leftarrow H \setminus \left\{\mathsf{esg}_g^{\mathsf{diff}}\right\}$
Output: $\{\mathsf{esg}_g^{\mathsf{rfn}}\}_g$

Algorithm 4. $\mathsf{connect}()$

Input: $R = \{\mathsf{esg}_g^{\mathsf{rfn}}\}_g$, egEcl_P, egErt_P
400: **for all** $\mathsf{esg}_g^{\mathsf{rfn}} \in R$ **do**
401: **for all** $w \in \mathsf{esgV}_g^{\mathsf{rfn}}$ **do**
402: let g' be the function to which w calls
403: **if** $\exists v : v \overset{\mathsf{ext}}{\approx} w$ **then**
404: **for all** $v' : (v, v') \in \mathsf{egEcl}_P$ **do**
405: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \cup \{(w, w')\}$ where $v' \overset{\mathsf{ext}}{\approx} w'$
406: **for all** $v'' : (v'', v) \in \mathsf{egErt}_P$ **do**
407: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \cup \{(w'', w)\}$ where $v'' \overset{\mathsf{ext}}{\approx} w''$
408: **else if** $\exists v : v \sim w \wedge v \overset{\mathsf{ext}}{\not\approx} w$ **then**
409: **for all** $v' : (v, v') \in \mathsf{egEcl}_P$ **do**
410: **if** $\exists w' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}} : v' \sim w' \ \wedge w'$ is an entry call node **then**
411: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \cup \{(w, w')\}$
412: **else**
413: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \ \cup \{(w, w') \mid w' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an entry call node$\}$
414: **for all** $v'' : (v'', v) \in \mathsf{egErt}_P$ **do**
415: **if** $\exists w'' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}} : v'' \sim w'' \ \wedge w''$ is an exit call node **then**
416: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \cup \{(w'', w)\}$
417: **else**
418: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \ \cup \{(w'', w) \mid w'' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an exit call node$\}$
419: **else**
420: $\mathsf{egEcl}_Q \leftarrow \mathsf{egEcl}_Q \ \cup \{(w, w') \mid w' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an entry call node$\}$
421: $\mathsf{egErt}_Q \leftarrow \mathsf{egErt}_Q \ \cup \{(w'', w) \mid w'' \in \mathsf{esgV}_{g'}^{\mathsf{rfn}}$ is an exit call node$\}$
Output: eg_Q

Algorithm 3 refine() uses the system call behavior of each called function to determine if any cross edges should be removed and others used in their places. (Analysis in Algorithm 2 does not account for the behavior of called functions when adding edges.)

Finally, Algorithm 4 connect() tries to copy call edges and return edges from the execution graph of the old program when we have sufficient matching support (line 405 and 407). Otherwise, we build call and return edges based on static analysis (lines 411, 413, 416, 418, 420, and 421).

B Proofs

Proof of Lemma 1. Since $v \overset{\mathsf{ext}}{\approx} w$, by Definition 1, 2, 3, for a call cycle $\langle v, v_2, \ldots, v_n, v\rangle$ in cfg_P, there will be a call cycle $\langle w, w_2, \ldots, w_n, w\rangle$ in cfg_Q such that $v_i \sim w_i : i \in [2, n]$, and if v_i and w_i are system call nodes, they must make the same system call, so these two call cycles result in the same (possibly empty) sequence of system calls. □

Proof of Lemma 2. If (w, w') is added to $\mathsf{esgE}_g^{\mathsf{cp}}$ in line 104, then consider the cross edge $(v, v') \in \mathsf{esgE}_g$ chosen in line 100. Since $(v, v') \in \mathsf{esgE}_g$, there is a full, silent path $p' = \langle v, \ldots, v'\rangle$ in P that was exercised in training. Consider the pruned path p from v to v' obtained by collapsing each call cycle in p' to its function call node. By

line 103, there is a corresponding $q \in \mathsf{isg}_g$ on which every node is extended-similar to its corresponding one in p (and hence $p \in \mathsf{isg}_f$, as well). Then, by Lemma 1 there is a full path q' that strongly supports (w, w'). □

Proof of Lemma 3. If an edge (w_i, w_j) is added to $\mathsf{esgE}_g^{\mathsf{diff}}$ at line 206, then w_i and w_j are call nodes with no call node in between them on q. As such, $\langle w_i, \ldots, w_j \rangle$ is a full, silent path that supports (w_i, w_j). □

Proof of Lemma 4. We first argue that any $(w', w'') \in \mathsf{esgE}_g^{\mathsf{rfn}}$ at the completion of $\mathsf{refine}()$ is supported by a full path. First, if (w', w'') was added to $\mathsf{esgE}_g^{\mathsf{cp}}$ in line 104 and then copied forward (lines 200, 302), or if (w', w'') was added to $\mathsf{esgE}_g^{\mathsf{diff}}$ in line 206 and then copied forward (line 302), then (w', w'') is supported by a full path per Lemmas 2 and 3. Now, suppose that (w', w'') was added in line 308 or 318. Then line 305 (respectively, 316) says that some call cycle from w is silent. So, if the cross edges (w', w), (w, w'') were supported by full paths, then the new cross edge (w', w'') is also supported by a full path. It follows by induction, with Lemmas 2–3 providing the base cases, that any cross edges added in lines 308 and 318 are supported by a full path.

We now show that any such edge is strongly supported. Consider any function call node $w \in \mathsf{esgV}_g^{\mathsf{rfn}}$ at the completion of refine. If $w \in \mathsf{esgV}_g^{\mathsf{cp}}$, then it was added in line 102 because it matched some v (line 101) from which an audible call cycle was traversed during training of eg_P. If $v \overset{\mathsf{ext}}{\approx} w$, then by Lemma 1, there is an audible call cycle from w, as well. If $v \overset{\mathsf{ext}}{\not\approx} w$ or $w \notin \mathsf{esgV}_g^{\mathsf{cp}}$, then w satisfied the condition in line 303 and, if there is no audible call cycle from w, was removed in lines 306–312. □

Proof of Lemma 5. Consider an edge added in line 405. Since both v and v' were witnessed during training eg_P, each is a system call node or has some audible call cycle. Because $v \overset{\mathsf{ext}}{\approx} w$ and $v' \overset{\mathsf{ext}}{\approx} w'$, Lemma 1 implies that each of w and w' is a system call node or has some audible call cycle. Moreover, Lemma 1 guarantees that w' is an entry call node since v' is, and so the call edge (w, w') created at line 405 is strongly supported by a full path. By similar reasoning, each return edge added at line 407 is strongly supported by a full path.

In all other cases in which an edge (w, w') is added to egEcl_Q (in line 411, 413, or 420), $\mathsf{connect}()$ explicitly checks whether w' is an entry call node for the function g' called by w (line 402), and so there is a full path supporting (w, w'). Similarly, for each edge (w'', w) added to egErt_Q, there is a full path supporting this edge. Since all nodes in each $\mathsf{esgV}_g^{\mathsf{rfn}}$ are either system call nodes or function call nodes from which there is an audible call cycle, these edges are strongly supported. □

C Training

In this appendix we briefly explain how we collected the traces for the four case studies, since training plays an important role in building the execution graphs. For ncompress and unzip, we tried all operation types and options listed in the online manuals. However, for tar and ProFTPD, we did not train as exhaustively as we did for the previous two cases due to the complexity of tar operations and ProFTPD configurations. Nevertheless, for

tar and ProFTPD, we did follow guidelines to enhance the repeatability of the training procedure, as described below.

tar. Following the manual (see `http://www.gnu.org/software/tar/manual/tar.pdf`), we trained tar for its three most frequently used operations (*create*, *list* and *extract*) that are introduced in Chapter 2 and with all options described in Chapter 3. The directory and files we adopted for applying those operations were the downloaded source of tar-1.14.

ncompress. We trained ncompress on its own source directory for version 4.2.4, using all operations and options described in its online manual (see `http://linux.about.com/od/commands/a/blcmdl1_compres.htm`),

ProFTPD. We trained ProFTPD configured using the sample configuration file shipped with the source, and with all commands described in the online manual (see `http://linux.about.com/od/commands/l/blcmdl1_ftp.htm`). We chose to transfer files within the ProFTPD-1.3.0 source directory.

unzip. Similar to the training on ncompress, we followed the unzip online manual (see `http://linux.about.com/od/commands/l/blcmdl1_unzip.htm`) and trained the program on the .zip package of version 5.52.

Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration

Juan Caballero[2,1], Zhenkai Liang[3], Pongsin Poosankam[2,1], and Dawn Song[1]

[1] UC Berkeley
[2] Carnegie Mellon University
[3] National University of Singapore

Abstract. Signature-based input filtering is an important and widely deployed defense. But current signature generation methods have limited coverage and the generated signatures often can be easily evaded by an attacker with small variations of the exploit message. In this paper, we propose *protocol-level constraint-guided exploration*, a new approach towards generating high coverage vulnerability-based signatures. In particular, our approach generates high coverage, yet compact, *vulnerability point reachability predicates*, which capture many paths to the vulnerability point. In our experimental results, our tool, *Elcano*, generates compact, high coverage signatures for real-world vulnerabilities.

1 Introduction

Automatic signature generation remains an important open problem. According to Symantec's latest Internet Security Threat Report hundreds of new security-critical vulnerabilities were discovered in the second half of 2007 [1]. For many of these vulnerabilities, the exploit development time is less than a day, while the patch development time is often days or months [1]. In addition, the patch deployment time can be long due to extensive testing cycles.

To address these issues, *signature-based input filtering* has been widely deployed in Intrusion Prevention (IPS) and Intrusion Detection (IDS) systems. Signature-based input filtering matches program inputs against a set of signatures and flags matched inputs as attacks. It provides an important means to protect vulnerable hosts when patches are not yet available or have not yet been applied. Furthermore, for legacy systems where patches are no longer provided by the vendor, or critical systems where any changes to the code might require a lengthy re-certification process, signature-based input filtering is often the only practical solution to protect the vulnerable program.

The key technical challenge to effective signature-based defense is to automatically and quickly generate signatures that have both low false positives and low false negatives. In addition, it is desirable to be able to generate signatures without access to the source code. This is crucial to wide deployment since it enables third-parties to generate signatures for commercial-off-the-shelf (COTS) programs, without relying on software vendors, thus enabling a quick response to newly found vulnerabilities.

Due to the importance of the problem, many different approaches for automatic signature generation have been proposed. Early work proposed to generate *exploit-based*

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 161–181, 2009.
© Springer-Verlag Berlin Heidelberg 2009

signatures using patterns that appeared in the observed exploits, but such signatures can have high false positive and negative rates [2,3,4,5,6,7,8,9,10]. More recently, researchers proposed to generate *vulnerability-based signatures*, which are generated by analyzing the vulnerable program and its execution and the actual conditions needed to exploit the vulnerability and can guarantee a zero false positive rate [11,12].

Automatic vulnerability signature generation. A vulnerability is a point in a program where execution might "go wrong". We call this point the *vulnerability point*. A vulnerability is only exploited when a certain condition, the *vulnerability condition*, holds on the program state when the vulnerability point is reached. Thus, to exploit a vulnerability, the input needs to satisfy two conditions: (1) it needs to lead the program execution to reach the vulnerability point; (2) the program state needs to satisfy the vulnerability condition at the vulnerability point. We call the condition that denotes whether an input message will make the program execution reach the vulnerability point the *vulnerability point reachability predicate* (VPRP). Thus, the problem of automatically generating a vulnerability-based signature can be decomposed into two: identifying the vulnerability condition and identifying the vulnerability point reachability predicate. A vulnerability-based signature is simply the conjunction of the two. While both problems are important, the space limitations makes trying to cover both in a single paper unrealistic. Thus, in this paper we focus on how to generate vulnerability point reachability predicates with high coverage and compact size, and we refer the reader to [13] for details on the vulnerability condition extraction. In this paper, we use *optimal signature* to refer to a vulnerability signature that has no false positives and no false negatives.

Coverage is a key challenge. One important problem with early vulnerability-based signature generation approaches [11, 12] is that the signatures only capture a single path to the vulnerability point (i.e., their VPRP contains only one path). However, the number of paths leading to the vulnerability point can be very large, sometimes infinite. Thus, such signatures are easy to evade by an attacker with small modifications of the original exploit message, such as changing the size of variable-length fields, changing the relative ordering of the fields (e.g., HTTP), or changing field values that drive the program through a different path to the vulnerability point [14,15].

Acknowledging the importance of enhancing the coverage of vulnerability-based signatures, recent work tries to incorporate multiple paths into the VPRP either by static analysis [16], or by dynamic analysis [17,18]. However, performing precise static analysis on binaries is hard due to issues such as indirection, pointers and loops.

ShieldGen takes a probing-based approach using protocol format information [18]—using the given protocol format, it generates different well-formed variants of the original exploit using various heuristics and then checks whether any of the variants still exploits the vulnerability. The advantage of this approach is that by using protocol format information, the final signature is expressed at the protocol level (which we call *protocol-level* signature) instead of the byte level. Compared to signatures at the byte-level (which do not understand the protocol format), protocol-level signatures have two advantages: they are more compact and they naturally cover variants of the exploits caused by variable-length fields and field re-ordering (See more detail in Section 2.2). The disadvantage of the approach used by ShieldGen is that the exploration uses heuristics to figure out what

test inputs to generate. Such heuristics can introduce false positives and do not use the information from the execution of the program, which would increase the coverage of the program execution space. As a result, the exploration is inefficient and has various limitations (See Section 2.3).

Bouncer extends previous approaches using symbolic execution to generate symbolic constraints on inputs as signatures [17]. Even though Bouncer makes improvements in increasing the coverage of the generated signatures, it still suffers from several limitations. First, it generates byte-level signatures instead of protocol-level signatures. As a result, it is difficult for Bouncer to handle evasion attacks using variable-length fields and field re-ordering. Second, Bouncer's exploration is inefficient and largely heuristic-based. As mentioned in their paper, the authors tried to use symbolic-constraint-guided exploration to explore the program execution space to identify different paths reaching the vulnerability point, but couldn't make the approach scale to real-world programs and thus had to resort to heuristics such as duplicating or removing parts of the input message or sampling certain field values to try to discover new paths leading to the vulnerability point. Thus, a key open problem for generating accurate and efficient signatures is how to generate vulnerability point reachability predicates with high coverage.

Our approach. In this paper, we propose *protocol-level constraint-guided exploration*, a new approach to automatically generate vulnerability point reachability predicates with high coverage, for a given vulnerability point and an initial exploit message. Our approach has 3 main characteristics: 1) it is *constraint-guided* (i.e., instead of heuristics-based exploration as in ShieldGen and Bouncer), 2) the constraint-guided exploration works at the *protocol-level* and generates protocol-level signatures at the end, and 3) it effectively *merges* explored execution paths to remove redundant exploration. The three points seamlessly weave together and amplify each other's benefit. By using constraint-guided exploration, our approach significantly increases the effectiveness and efficiency of the program execution space exploration. By lifting the symbolic constraints from the byte level to the protocol level, our constraint-guided exploration is done at the protocol level, which makes the exploration feasible for real-world programs, addressing the problem that Bouncer couldn't solve. By merging paths in the exploration, we further reduce the exploration space.

Elcano. We have designed and developed *Elcano*, realizing the aforementioned approach. We have evaluated the effectiveness of our system using real-world vulnerable programs. In our experiments, Elcano achieved optimal or close-to-optimal results in terms of coverage. In addition, the generated signatures are compact. In fact, most of the signatures are so compact that they can be understood by a human.

Compared to Bouncer, Elcano produces higher coverage signatures. For example, for the GHttpd vulnerability Bouncer run for 24 hours, exploring only some fraction of all possible paths, and produced a partial signature with significant false negatives. In contrast, Elcano generates an optimal signature for the same vulnerability in 55 seconds. Compared to ShieldGen, Elcano produces more accurate signatures, both in terms of less false negatives (i.e., higher coverage) and less false positives.

In addition to signature generation, extracting a high coverage vulnerability point reachability predicate is useful for other applications such as exploit generation [19] and patch testing. For example, the Microsoft patch MS05-018 missed some paths to the vulnerability point and as a result left the vulnerability still exploitable after the patch [20]. This situation is not uncommon. A quick search on the CVE database returns 13 vulnerabilities that were incorrectly or incompletely patched [21]. Our technique could assist software developers to build more accurate patches. Furthermore, our protocol-level constraint-guided approach can increase the effectiveness of generating high-coverage test cases and hence be very valuable to software testing and bug finding.

2 Problem Definition and Approach Overview

In this section, we first introduce the problem of automatic generation of protocol-level vulnerability point reachability predicates, then present our running example and finally give the overview of our approach.

2.1 Problem Definition

Automatic generation of protocol-level vulnerability point reachability predicates. Given a parser implementing a given protocol specification, the vulnerability point, and an input that exploits the vulnerability at the vulnerability point in a program, the problem of automatic generation of protocol-level vulnerability point reachability predicates is to automatically generate a predicate function F, such that when given some input mapped into field structures by the parser, F evaluates over the field structures of the input: if it evaluates to *true*, then the input is considered to be able to reach the vulnerability point, otherwise it is not.

Parser availability and specification quality. The problem of automatic generation of protocol-level vulnerability point reachability predicates assumes the availability of a parser implementing a given protocol or file specification. Such requirement is identical to previous approaches such as ShieldGen [18]. The parser given some input data can map it into fields, according to the specification, or fail if the input is malformed. In the latter case, the IDS/IPS could opt to block the input or let it go through while logging the event or sending a warning. Such parser is available for common protocols (e.g., Wireshark [22]), and many commercial network-based IDS or IPS have such a parser built-in. In addition, recent work has shown how to create a generic parser that takes as input multiple protocol specifications written in an intermediate language [23,24].

The quality of the specification used by the parser matters. While obtaining a high quality specification is not easy, this is a one time effort, which can be reused for multiple signatures, as well as other applications. For example, in our experiments we extracted a WMF file format specification. According to the CVE Database [21] the WMF file format appears in 21 vulnerabilities, where our specification could be reused. Similarly, an HTTP specification could be reused in over 1500 vulnerabilities. Also, recent work has proposed to automatically extract the protocol specification from the program binary [25,26,27,28]. Such work can be used when the protocol used by the vulnerable program has no public specification.

```
1  void service() {
2   char msgBuf[4096];
3   char lineBuf[4096];
4   int nb=0, i=0, sockfd=0;
5   nb=recv(sockfd,msgBuf,4096,0);
6   for(i = 0; i < nb; i++) {
7     if (msgBuf[i] == '\n')
8         break;
9     else
10      lineBuf[i] = msgBuf[i];
11  }
12  if (lineBuf[i-1] == '\r')
13    lineBuf[i-1] = '\0'
14  else lineBuf[i] = '\0';
15  doRequest(lineBuf);
16 }
```

```
17 void doRequest(char *lineBuf){
18  char vulBuf[128],uri[256];
19  char ver[256], method[256];
20  int is_cgi = 0;
21  sscanf(lineBuf,
22    "%255s %255s %255s",
23    method, uri, ver);
24  if (strcmp(method,"GET")==0 ||
25      strcmp(method,"HEAD")==0){
26    if strncmp(uri,"/cgi-bin/",
27      9)==0  is_cgi = 1;
28    else is_cgi = 0;
29    if (uri[0] != '/') return;
30    strcpy(vulBuf, uri);
31  }
32 }
```

Fig. 1. Our running example

Exploit availability. Similarly to all previous work on automatic generation of vulnerability signatures [12,11,17,18], our problem definition assumes that an initial exploit message is given.

Vulnerability point availability. Finally, our problem definition assumes that the vulnerability point is given. Identifying the vulnerability point is part of a parallel project that aims to accurately describe the vulnerability condition [13]. Such vulnerability point could also be identified using previous techniques [17,29].

2.2 Running Example

Figure 1 shows our running example. We represent the example in C language for clarity, but our approach operates directly on program binaries. Our example represents a basic HTTP server and contains a buffer-overflow vulnerability. In the example, the `service` function copies one line of data received over the network into `linebuf` and passes it to the `doRequest` function that parses it into several field variables (lines 21-23) and performs some checks on the field values (lines 24-31). The first line in the exploit message includes the method, the URI of the requested resource, and the protocol version. If the method is GET or HEAD (lines 24-25), and the first character of the URI is a slash (line 29), then the vulnerability point is reached at line 30, where the size of `vulBuf` is not checked by the `strcpy` function. Thus, a long URI can overflow the `vulBuf` buffer.

In this example, the vulnerability point is at line 30, and the vulnerability condition is that the local variable `vulBuf` will be overflowed if the size of the URI field in the received message is greater than 127. Therefore, for this example, the vulnerability point reachability predicate is: `(strcmp(FIELD_METHOD,"GET") == 0 || strcmp(FIELD_METHOD,"HEAD") == 0) && FIELD_URI[0]` $\neq$ `'/'` while the vulnerability condition is: `length(FIELD_URI) > 127`, and the conjunction of the two is an optimal protocol-level signature.

2.3 Approach

In this paper we propose a new approach to generate high coverage, yet compact, vulnerability point reachability predicates, called *protocol-level constraint-guided exploration*. Next, we give the motivation and an overview of the three characteristics that comprise our approach.

Constraint-guided. As mentioned in Section 1, previous approaches such as ShieldGen and Bouncer use heuristics-based exploration [17,18]. Heuristic-based exploration suffers from a fundamental limitation: the number of probes needed to exhaustively search the whole space is usually astronomical. In addition, an exhaustive search is inefficient as many probes end up executing the same path in the program. Thus, such approaches often rely on heuristics that are not guaranteed to significantly increase the signature's coverage and can also introduce false positives.

For example, ShieldGen [18] first assumes that fields can be probed independently, and then for fixed-length fields it samples just a few values of each field, checking whether the vulnerability point is reached or not for those values. Probing each field independently means that conditions involving multiple fields cannot be found. Take the condition `SIZE1 + SIZE2` $\leq$ `MSG_SIZE`, where `SIZE1` and `SIZE2` are length fields in the input, and `MSG_SIZE` represents the total length of the received message. The authors of ShieldGen acknowledge that their signatures cannot capture this type of conditions, but such conditions are commonly used by programs to verify that the input message is well-formed and failing to identify them will introduce either false positives or false negatives, depending on the particular heuristic. Probing only a few sample values for each field is likely to miss constraints that are satisfied by only a small fraction of the field values. For example, a conditional statement such as `if (FIELD==10) || (FIELD==20) then exploit, else safe`, where FIELD is a 32-bit integer, creates two paths to the vulnerability point. Finding each of these paths would require 2^{30} random probes on average to discover. Creating a signature that covers both paths is critical since if the signature only covers one path (e.g., `FIELD == 10`), the attacker could easily evade detection by changing FIELD to have value 20.

To overcome these limitations, we propose to use a constraint-guided approach by monitoring the program execution, performing symbolic execution to generate path predicates, and generating new inputs that will go down a different path. This constraint-guided exploration is similar in spirit to recent work on using symbolic execution for automatic test case generation [30,31,32]. However, simply applying those techniques does not scale to real-world programs, given the exponential number of paths to explore. In fact, in Bouncer [17] the authors acknowledge that they wanted to use a constraint-guided approach but failed to do so due to the large number of paths that need to be explored and thus had to fall back to the heuristics-based probing approach.

To make the constraint-guided exploration feasible and effective we have incorporated two other key characteristics into our approach as described below.

Protocol-level constraints. Previous symbolic execution approaches generate what we call *stream-level conditions*, i.e., constraints that are evaluated directly on the stream of input bytes. Such stream-level conditions in turn generate *stream-level signatures*, which are also specified at the byte level. However, previous work has shown that

signatures are better specified at the protocol-level instead of the byte level [6, 18]. We call such signatures *protocol-level signatures*.

Our contribution here is to show that, by lifting stream-level conditions to *protocol-level conditions*, so that they operate on protocol fields rather than on the input bytes, we can make the constraint-guided approach feasible, as using constraints at the protocol-level hugely reduces the number of paths to be explored compared to using stream-level conditions. The state reduction is achieved in two ways. First, the parsing logic often introduces huge complexity in terms of the number of execution paths that need to be analyzed. For example, in our experiments, 99.8% of all constraints in the HTTP vulnerabilities are generated by the parsing logic. While such parsing constraints need to be present in the stream-level conditions, they can be removed in the protocol-level conditions. Second, the stream-level conditions introduced by the parsing logic fixes the field structure to be the same as in the original exploit message, for example fixing variable-length fields to have the same size as in the original exploit message, and fixing the field sequence to be the same as in the exploit message (when protocols such as HTTP allow fields to be reordered). Unless the parsing conditions are removed the resulting signature would be very easy to evade by an attacker by applying small variations to the field structure of the exploit message. Finally, the vulnerability point reachability predicates at the protocol level are smaller and easier to understand by humans.

Merging execution paths. The combination of protocol-level conditions with constraint-guided exploration is what we call *protocol-level constraint-guided exploration*, an iterative process that incrementally discovers new paths leading to the vulnerability point. Those paths need to be added to the vulnerability point reachability predicate. The simplistic approach would be to blindly explore new paths by reversing conditions and at the end create a vulnerability point reachability predicate that is a disjunction (i.e., an enumeration) of all the discovered paths leading to the vulnerability point. Such approach has two main problems. First, blindly reversing conditions produces a search space explosion, since the number of paths to explore becomes exponential in the number of conditions, and much larger than the real number of paths that exist in the program. We explain this in detail in Section 4. In addition, merely enumerating the discovered paths generates signatures that quickly explode in size.

To overcome those limitations, we utilize the observation that the program execution may fork at one condition into different paths for one processing task, and then merge back to perform another task. For example, a task can be a validation check on the input data. Each independent validation check may generate one or multiple new paths (e.g., looking for a substring in the HTTP URL generates many paths), but if the check is passed then the program moves on to the next task, which usually merges the execution back into the original path. Thus, in our exploration, we use a *protocol-level exploration graph* to identify such potential merging points. This helps alleviate the search space explosion problem, and allows our exploration to quickly reach high coverage.

2.4 Architecture Overview

We have implemented our approach in a system called Elcano. The architecture of Elcano is shown in Figure 2. It comprises of two main components: the *constraint*

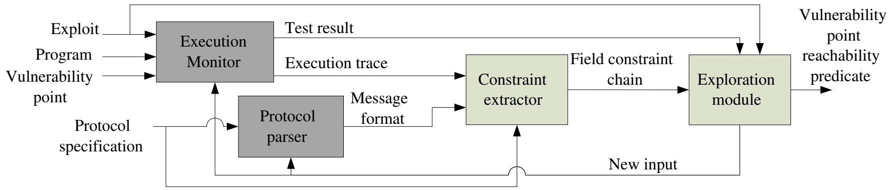


Fig. 2. Elcano architecture overview. The darker color modules are given, while the lighter color components have been designed and implemented in this work.

extractor and the *exploration module*, and two off-the-shelf assisting components: the *execution monitor* and *the parser*.

The overall exploration process is an iterative process that incrementally explores new execution paths. In each iteration (that we also call test), an input is sent to the program under analysis, running inside the execution monitor. The execution monitor produces an execution trace that captures the complete execution of the program on the given input, including which instructions were executed and the operands content. The execution monitor also logs the test result, i.e., whether the vulnerability point was reached or not during the execution. In addition, the parser extracts the message format for the input, according to the given protocol specification.

Then, given the execution trace and the message format, the constraint extractor obtains the *field constraint chain*. The field constraint chain is conceptually similar to the *path predicate* used in previous work on automatic test case generation, but the conditions are at the protocol-level and each condition is tagged with additional information. We detail the field constraint chain and its construction in Section 3.

The exploration module maintains the *protocol-level exploration graph*, which stores the current state of the exploration, i.e., all the execution paths that have been so far explored. Given the field constraint chain, the exploit message and the test result, the exploration module merges the new field constraint chain into the current protocol-level exploration graph. Then, the exploration module uses the protocol-level exploration graph to select a new path to be explored and generates a new input that will lead the program execution to traverse that path. Given the newly generated input, another iteration begins. We detail the exploration module in Section 4.

The process is started with the initial exploit message and runs iteratively until there are no more paths to explore or a user-specified time-limit is reached. At that point the exploration module outputs the VPRP. The VPRPs produced by Elcano are written using the Vine language [33] with some extensions for string operations [34]. The Vine language is part of the Bitblaze binary analysis infrastructure [35].

3 Extracting the Protocol-Level Path-Predicate

In this section we present the constraint extractor, which given an execution trace, produces a field constraint chain. The architecture of the constraint extractor is shown in Figure 3. First, given the execution trace the *path predicate extractor* performs symbolic execution with the input represented as a symbolic variable and extracts the *path predicate*, which is essentially the conjunction of all branch conditions dependent on the

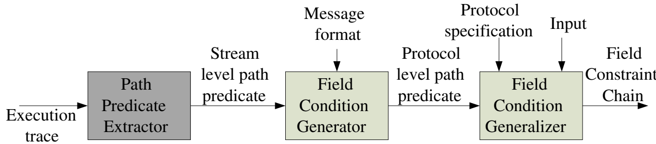


Fig. 3. Constraint Extractor Architecture. The darker color module is given, while the lighter color components have been designed and implemented in this work.

symbolic input in the execution captured in the execution trace. The concept of symbolic execution, the path predicate and how to compute it are well understood and have been widely used in previous work including vulnerability signature generation [11,12] and automatic test case generation [30,31]. Thus, we refer the interested reader to these previous work for details.

The path predicate generated by previous work is at the stream-level, i.e., the conditions are on raw bytes of the input. To enable constraint-guided exploration, Elcano needs to lift the path predicate from the stream-level to the protocol-level, where the conditions are instead on field variables of the input. To make the distinction clear, we refer to the path predicate at the stream-level the *stream-level path-predicate*, and the path predicate at the protocol-level the *protocol-level path-predicate*. In addition, the constraint extractor needs to remove the parsing conditions, which dramatically reduces the exploration space and makes the constraint-guided exploration feasible.

To accomplish this, first the *field condition generator* lifts the stream-level path-predicate to the protocol-level, and then the *field condition generalizer* generalizes it by removing the parsing conditions and outputs the *field constraint chain*, which is essentially the protocol-level path-predicate, where each condition is annotated with some additional information and conditions are ordered using the same order as they appeared in the execution.

3.1 The Field Condition Generator

Given the stream-level path-predicate generated by the path predicate extractor and the message format of the input given by the parser, the field condition generator outputs a protocol-level path-predicate. It performs this in two steps. First, it translates each byte symbol `INPUT[x]` in the stream-level path-predicate into a field symbol `FIELD_fieldname [x - start(fieldname)]` using the mapping produced by the parser. Second, it tries to combine symbols on consecutive bytes of the same field. For example, the stream-level path-predicate might include the following condition: `(INPUT[6] << 8 | INPUT[7]) == 0`. If the message format states that inputs 6 and 7 belong to the same 16-bit `ID` field, then the condition first gets translated to `(FIELD_ID[0] << 8 | FIELD_ID[1]) == 0` and then it is converted to `FIELD_ID == 0` where `FIELD_ID` is a 16-bit field symbol.

The message format provided by the parser is a hierarchical tree, where one field may contain different subfields, with the root of the tree representing the whole message. For example, the `linebuf` variable in our running example represents the

Request-Line field, which in turn contains 3 subfields: Method, Request-URI, and HTTP-Version. Thus, a condition such as: strstr(linebuf,"../") $\neq$ 0 would be translated as strstr(FIELD_Request-Line,"../") $\neq$ 0. A condition on the whole message would translate into a condition on the special MSG field.

Benefits. This step lifts the stream-level path-predicate to the protocol-level, breaking the artificial constraints that the stream-level path-predicate imposes on the position of fields inside the exploit message. For example, protocols such as HTTP allow some fields in a message (i.e., all except the Request-Line/Status-Line) to be ordered differently without changing the meaning of the message. Thus, two equivalent exploit messages could have the same fields ordered differently and a byte-level vulnerability point reachability predicate generated from one of them would not flag that the other also reaches the vulnerability point. In addition, if variable-length fields are present in the exploit message, changing the size of such fields changes the position of all fields that come behind it in the exploit message. Again, such trivial variation of the exploit message could defeat stream-level signatures. Thus, by expressing constraints using field symbols, protocol-level signatures naturally allow a field to move its position in the input.

3.2 The Field Condition Generalizer

The field condition generalizer takes as input the protocol-level path-predicate generated by the field condition generator, the protocol specification and the input that was sent to the program and outputs a field constraint chain where the parsing-related conditions have been removed.

First, the field condition generalizer assigns a symbolic variable to each byte of the input and processes the input according to the given protocol specification. This step generates symbolic conditions that capture the constraints on the input which restrict the message format of the input to be the same as the message format returned by the parser on the given input. We term these conditions the parsing conditions. Then, the field condition generalizer removes the parsing conditions from the protocol-level path-predicate by using a fast syntactic equivalence check. If the fast syntactic check fails, the field condition generalizer uses a more expensive equivalence check that uses a decision procedure.

Benefits. The parsing conditions in the protocol-level path-predicate over-constrain the variable-length fields, forcing them to have some specific size (e.g., the same as in the exploit message). Thus, removing the parsing conditions allows the vulnerability point reachability predicate to handle exploit messages where the variable-length fields have a size different than in the original exploit message. In addition, for some protocols such as HTTP, the number of parsing conditions in a single protocol-level path-predicate can range from several hundreds to a few thousands. Such a huge number of unnecessary conditions would blow up the size of the vulnerability point reachability predicate and negatively impact the exploration that we will present in Section 4. Note that the parsing conditions are enforced by the parser, so we can safely remove them from the protocol-level path-predicate while still having the conditions enforced during the signature matching time. We refer the reader to the extended version for more details [36].

The field constraint chain. To assist the construction of the protocol-level exploration graph (explained in Section 4), the constraint extractor constructs the *field constraint chain* using the generalized protocol-level path-predicate (after the parsing conditions have been removed). A field constraint chain is an enhanced version of the protocol-level path-predicate where each branch condition is annotated with the instruction counter and an MD5 hash of the callstack of the program at the branching point, and these annotated branch conditions are put in an ordered chain using the same order as they appear in the execution path.

4 Execution-Guided Exploration

In this section we present the exploration module, which adds the given field constraint chain to the protocol-level exploration graph, selects a new path to be explored and generates an input that will traverse that path. That input is used to start a new iteration of the whole process by sending it to the program running in the execution monitor. Once there are no more paths to explore or a user-specified time-limit is reached, the exploration module stops the exploration and outputs the VPRP.

Our exploration is based on a *protocol-level exploration graph*, which makes it significantly different from the traditional constraint-based exploration used in automatic test case generation approaches [30, 31, 37]. Using a protocol-level exploration graph provides two fundamental benefits: 1) the exploration space is significantly reduced, and 2) it becomes easy to merge paths, which in turn further reduces the exploration space, and reduces the size of the vulnerability point reachability predicate. In this section, we first introduce the protocol-level exploration graph, next we present our intuition for merging paths, and then we describe the exploration process used to extract the vulnerability point reachability predicate.

4.1 The Protocol-Level Exploration Graph

The explorer dynamically builds a *protocol-level exploration graph* as the exploration progresses. In the graph, each node represents an input-dependant branching point (i.e., a conditional jump) in the execution, which comprises the protocol-level condition and some additional information about the state of the program when the branching point was reached, which we explain in Section 4.2. Each node can have two edges representing the branch taken if the node's condition evaluated to true (`T`) or false (`F`). We call the node where the edge originates the *source node* and the node where the edge terminates the *destination node*. If a node has an *open edge* (i.e, one edge is missing), it means that the corresponding branch has not yet been explored.

4.2 Merging Execution Paths

When a new field constraint chain is added to the protocol-level exploration graph, it is important to merge all conditions in the field constraint chain that are already present in the graph. Failure to merge a condition creates a duplicate node, which in turn effectively doubles the exploration space because all the subtree hanging from the replicated

node would need to be explored as well. Thus, as the number of duplicated nodes increases, the exploration space increases exponentially.

The key intuition behind why merging is necessary is that it is common for new paths generated by taking a different branch at one node, to quickly merge back into the original path. This happens because programs may fork execution at one condition for one processing task, and then merge back to perform another task. One task could be a validation check on the input data. Each independent check may generate one or multiple new paths (e.g., looking for a substring in the URI generates many paths), but if the check is passed then the program moves on to the next task (e.g., another validation check), which usually merges the execution back into the original path. For example, when parsing a message the program needs to determine if the message is valid or not. Thus, it will perform a series of independent validity checks to verify the values of the different fields in the message. As long as checks are passed, the program still considers the message to be valid and the execution will merge back into the original path. But, if a check fails then the program will move into a very different path, for example sending an error message.

The intuition on the merging is that two nodes can be merged if they represent the same program point and they are reached with the same program state. To identify the program point, each condition in the field constraint chain is annotated with the program's instruction counter (*eip*) and an MD5 hash of the callstack, both taken at the time the condition was executed. To identify the program state we use a technique similar to the one introduced in [38] where we compute the set of all values (both concrete and symbolic) written by the program during the execution up to the point where the condition is executed. Thus, we merge nodes that satisfy 4 conditions: same eip, same callstack hash, equivalent conditions, and same program state, where Elcano queries the decision procedure to determine if two conditions are equivalent.

4.3 The Exploration Process

Figure 4 shows the architecture of the exploration module. It is comprised of three components: the *explorer*, the *prioritization engine*, and the *input generator*, plus an off-the-shelf *decision procedure*. The exploration process is comprised of 3 steps: (1) given the field constraint chain, the explorer adds it to the current protocol-level exploration graph producing an updated graph; (2) given the updated protocol-level exploration graph, the prioritization engine decides which node's open edge to explore next; (3) for

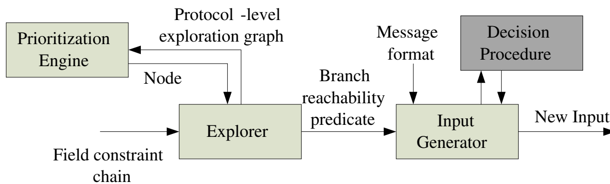


Fig. 4. Exploration module architecture. The darker color module is given, while the lighter color components have been designed and implemented in this work.

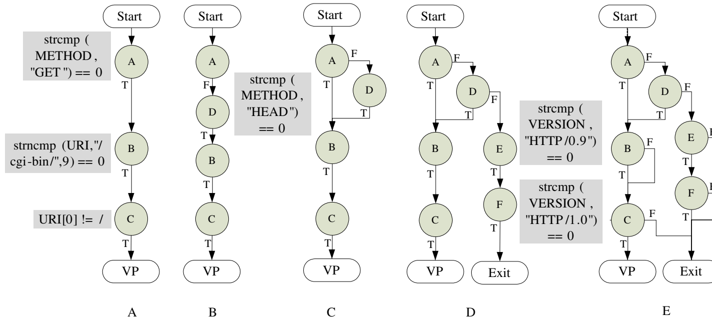


Fig. 5. Building the protocol-level exploration graph for our running example

the selected node's open edge, the *input generator* generates a new input that will lead the program execution to reach that node and follow the selected open edge.

The new input is then used to start another iteration of the whole process as shown in Figure 2, that is, the new input is replayed to the program running in the execution monitor and a new field constraint chain is generated by the constraint extractor, which is passed to the explorer and so on. The prioritization engine is in charge of stopping the whole process once there are no more paths to explore or a user-specified time-limit is reached. When the exploration stops, the explorer outputs the VPRP.

Next, we detail the 3 steps in the exploration process and how to extract the VPRP. We illustrate the different steps using Figure 5 which represents the graph for our running example. Note that, the A–F node labels are not really part of the protocol-level exploration graph but we add them here to make it easier to refer to the nodes.

Adding the new path to the exploration graph. To insert a new field constraint chain into the protocol-level exploration graph, the explorer starts merging from the top until it finds a node that it cannot merge, either because it is not in the graph yet, or because the successor in the new field constraint chain is not the same one as in the graph. To check if the node is already in the graph, the explorer checks if the node to be inserted is equivalent (same EIP, same callstack hash, equivalent condition, and same state) to any other node already in the graph. We call the last node that can be merged from the top the *split node*.

Once a split node has been identified the explorer keeps trying to merge the rest of the nodes in the new field constraint chain until it finds a node that it can merge, which we term the *join node*. At that point, the explorer adds all the nodes in the new field constraint chain between the split node and the join node as a sequence of nodes in the graph hanging from the split node and merging at the join node. The process of looking for a split node and then for a join node is repeated until the sink of the new field constraint chain is reached. At that point, if the explorer was looking for a join node then all nodes between the last split node and the sink are added to the graph as a sequence that hangs from the last split node and ends at the sink.

For example, in Figure 5A the graph contains only the original field constraint chain generated by sending the starting exploit message to the program, which contains the

three nodes introduced by lines 24, 26, and 29 in our running example (since the parsing conditions have already been removed). The sink of the original field constraint chain is the vulnerability point node (VP). Figure 5B shows the second field constraint chain that is added to the graph, which was obtained by creating an input that traverses the false branch of node A. When adding the field constraint chain in Figure 5B to the graph in Figure 5A, the explorer merges node A and determines that A is a split node because A's successor in the new field constraint chain is not A's successor in the graph. Then, at node B the explorer finds a join node and adds node D between the split node and the join node in the graph. Finally node C is merged and we show the updated graph in Figure 5C.

Selecting the next node to explore. Even after removing the parsing conditions and merging nodes, the number of paths to explore can still be large. Since we are only interested in paths that reach the vulnerability point, we have implemented a simple prioritization scheme that favours paths that are more likely to reach it. The prioritization engine uses a simple weight scheme, where there are three weights 0, 1, and 2. Each weight has its own node queue and the prioritization engine always picks the first node from the highest weight non-empty queue. The explorer assigns the weights to the nodes when adding them to the graph. Nodes that represent loop exit conditions get a zero weight (i.e., lowest priority). Nodes in a field constraint chain that has the VP as sink get a weight of 2 (i.e., highest priority). All other nodes get a weight of 1. We favor nodes that are in a path to the VP because if a new path does not quickly lead back to the VP node, then the message probably failed the current check or went on to a different task and thus it is less likely to reach VP later. We disfavor loop exit conditions to delay unrolling the same loop multiple times. Such heuristic helps achieve high coverage quickly.

Generating a new input for a new branch. We define a *node reachability predicate* to be the predicate that summarizes how to reach a specific node in the protocol-level exploration graph from the Start node, which includes all paths in the graph from the Start to that node. Similarly, we define a *branch reachability predicate* to be the predicate that summarizes how to traverse a specific branch of a node. A branch reachability predicate is the conjunction of a node reachability predicate with the node's condition (to traverse the true branch), or the negation of the node's condition (to traverse the false branch). To compute a new input that traverses the specific branch selected by the prioritization engine, the explorer first computes the branch reachability predicate. Then, the input generator creates a new input that satisfies the branch reachability predicate.

To compute the branch reachability predicate, the explorer first computes the node reachability predicate. The node reachability predicate is essentially the weakest precondition (WP) [39] of the source node of the open edge over the protocol-level exploration graph—by definition, the WP captures all paths in the protocol-level exploration graph that reach the node. Then, the explorer computes the conjunction of the WP with the node's condition or with the negated condition depending on the selected branch. Such conjunction is the branch reachability predicate, which is passed to the input generator.

Table 1. Vulnerable programs used in the evaluation

Program	CVE	Protocol	Type	Guest OS	Vulnerability Type
gdi32.dll (v3159)	CVE-2008-1087	EMF file	Binary	Windows XP	Buffer overflow
gdi32.dll (v3099)	CVE-2007-3034	WMF file	Binary	Windows XP	Integer overflow
Windows DCOM RPC	CVE-2003-0352	RPC	Binary	Windows XP	Buffer overflow
GHttpd	CVE-2002-1904	HTTP	Text	Red Hat 7.3	Buffer overflow
AtpHttpd	CVE-2002-1816	HTTP	Text	Red Hat 7.3	Buffer overflow
Microsoft SQL Server	CVE-2002-0649	Proprietary	Binary	Windows 2000	Buffer overflow

For example, in Figure 5C if the prioritization engine selects the false branch of node D to be explored next, then the branch reachability predicate produced by the explorer would be: $\overline{A}$ && $\overline{D}$. Similarly, in Figure 5D if the prioritization engine selects the false branch of node B to be explored next, then the branch reachability predicate produced by the explorer would be: (A||($\overline{A}$ && D)) && $\overline{B}$.

The input generator generates a new input that satisfies the branch reachability predicate using a 3-step process. First, it uses a decision procedure to generate field values that satisfy the branch reachability predicate. If the decision procedure returns that no input can reach that branch, then the branch is connected to the `Unreachable` node. Second, it extracts the values for the remaining fields (not constrained by the decision procedure) from the original exploit message. Third, it checks the message format provided by the parser to identify any fields that need to be updated given the dependencies on the modified values (such as length or checksum fields). Using all the collected field values it generates a new input and passes it to the replay tool. We refer the reader to our extended version [36] for our handling of field conditions that depend on a memory read from a symbolic address.

Extracting the vulnerability point reachability predicate. Once the exploration ends, the protocol-level exploration graph contains all the discovered paths leading to the vulnerability point. To extract the VPRP from the graph the explorer computes the node reachability predicate for the VP node. For our running example, represented in Figure 5E the VPRP is: (A||($\overline{A}$ && D)) && C. Note that, a mere disjunction of all paths to the VP, would generate the following VPRP: (A && B && C)||($\overline{A}$ && D && B && C)||(A && $\overline{B}$ && C)||($\overline{A}$ && D && $\overline{B}$ && C). Thus, Elcano's VPRP is more compact using 4 conditions instead of 14.

5 Evaluation

In this section, we present the results of our evaluation. We first present the experiment setup, then the constraint extractor results and finally the exploration results.

Experiment setup. We evaluate Elcano using 6 vulnerable programs, summarized in Table 1. The table shows the program, the CVE identifier for the vulnerability [21], the protocol used by the vulnerable program, the protocol type (i.e., binary or text), the guest operating system used to run the vulnerable program, and the type of vulnerability.

Table 2. Constraint extractor results for the first test, including the number of conditions in the protocol-level path-predicate and the number of remaining conditions after parsing conditions have been removed

Program	Original	Non-parsing conditions
Gdi-emf	860	65
Gdi-wmf	4	4
DCOM RPC	535	521
GHttpd	2498	5
AtpHttpd	6034	10
SQL Server	2447	7

Table 3. Exploration results, including whether all open edges in the protocol-level exploration graph were explored and the number of conditions remaining in the vulnerability point reachability predicate

Program	All branches explored	VPRP
Gdi-emf	no	72
Gdi-wmf	yes	5
DCOM RPC	no	1651
GHttpd	yes	3
AtpHttpd	yes	10
SQL Server	yes	3

We select the vulnerabilities to cover file formats as well as network protocols, multiple operating systems, multiple vulnerability types, and both open-source and closed programs, where no source code is available. In addition, the older vulnerabilities (i.e., last four) are also selected because they have been analyzed in previous work, and this allows us to compare our system's results to previous ones.

5.1 Constraint Extractor Results

In this section we evaluate the effectiveness of the constraint extractor, in particular of the field condition generalizer, at removing the parsing conditions from the protocol-level path-predicate. For simplicity, we only show the results for the protocol-level path-predicate produced by the field condition generator from the execution trace generated by the original exploit. Note that, during exploration this process is repeated once per newly generated input. Table 2 summarizes the results. The *Original* column represents the number of input-dependent conditions in the protocol-level path-predicate and is used as the base for comparison. The *Non-parsing conditions* column shows the number of remaining conditions after removing the parsing conditions.

The removal of the parsing conditions is very successful in all experiments. Overall, in the four vulnerable programs that include variable-length strings (i.e., excluding Gdi-wmf and DCOM-RPC), the parsing conditions account for 92.4% to 99.8% of all conditions. For formats that include arrays, such as DCOM RPC, the number of parsing conditions is much smaller but it is important to remove such conditions because otherwise they constrain the array to have the same number of elements as in the exploit message. By removing the parsing conditions, each field constraint chain represents many program execution paths produced by modifying the format of the exploit message (e.g., extending variable-length fields or reordering fields). This dramatically decreases the exploration space making the constraint-guided exploration feasible.

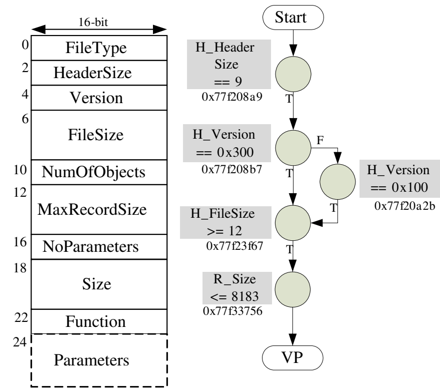
5.2 Exploration Results

Table 3 shows the results for the exploration phase. We set a user-defined time-limit of 6 hours for the exploration. If the exploration has not completed by that time Elcano

Table 4. Performance evaluation. The generation time and the average test time are given in seconds, and the trace size is given in Megabytes

Program	Gener. time	# tests	Ave. test time	Trace size
Gdi-emf	21600	502	43.0	28.8
Gdi-wmf	98	6	16.3	3.0
DCOM RPC	21600	235	92.0	3.5
GHttpd	55	6	9.1	3.0
AtpHttpd	282	12	23.5	8.6
SQL Server	1384	11	125.8	27.5

Table 5. On the left, the format of the Gdi-wmf exploit file. On the right the vulnerability point reachability predicate.



outputs the intermediate VPRP and stores the current state of the exploration. This state can later be loaded to continue the exploration at the same point where it was interrupted. The first column indicates whether the exploration completes before the specified time-limit. The second column presents the number of conditions in the intermediate VPRP that is output by the exploration module once there are no more paths to be explored or the time-limit is reached.

The results show that in 4 out of 6 experiments Elcano explored all possible paths, thus generating a complete VPRP. For the DCOM RPC and Gdi-emf experiments, the 6 hour time-limit was reached, thus the VPRPs are not complete. They also show that the number of conditions in the VPRP is in most cases small. The small number of conditions in the VPRP and the fact that in many cases those conditions are small themselves, makes the signatures easy for humans to analyze, as opposed to previous constraint-based approaches where the large number of conditions in the signature made it hard to gain insight on the quality of the signature. We do that by labeling the nodes in the graph with the full protocol-level conditions.

Performance. Table 4 summarizes the performance measurements for Elcano. All measurements were taken on a desktop computer with a 2.4GHz Intel Core2 Duo CPU and 4 GB of memory. The first column presents the VPRP generation time in seconds. For the Gdi-emf and DCOM RPC examples, the 6 hour time-limit on generation time is reached. For the rest, the generation time ranges from under one minute for the GHttpd vulnerability up to 23 minutes for the Microsoft SQL vulnerability. Most of the time (between 60% and 80% depending on the example) is spent by the constraint extractor. Thus, we plan to parallelize the exploration by having a central explorer, which spawns multiple copies of the constraint extractor and the execution monitor, each testing a different input and reporting back to the explorer. The remaining columns show the number of tests in the exploration, the average time per test in seconds, and the average size in Megabytes of the execution trace.

Compared to Bouncer, where the authors also analyze the SQL Server and GHttpd vulnerabilities, the signatures produced by Elcano have higher coverage (i.e., less false negatives) and are smaller. For example, Bouncer spends 4.7 hours to generate a signature for the SQL Server vulnerability, and the generated signature only covers a fraction of all the paths to the vulnerability point. In contrast, Elcano spends only 23 minutes, and the generated signature covers all input-dependnt branches to the vulnerability point. Similarly, for the GHttpd vulnerability the authors stop the signature generation after 24 hours, and again the signature only covers a fraction of all input-dependent branches to the vulnerability point, while Elcano generates a complete signature that covers all input-dependent branches to the vulnerability point in under one minute.

SQL server. The parser returns that there are two fields in the exploit message: the Command (CMD) and the Database name (DB). The original protocol-level path-predicate returned by the constraint extractor contains 7 conditions: 4 on the CMD field and the other 3 on the DB field. The exploration explores the open edges of those 7 nodes and finds that none of the newly generated inputs reaches the vulnerability point. Thus, no new paths are added to the graph and the VPRP is: `(FIELD_CMD==4)` && (strcmp(FIELD_DB,"")≠0)&& (strcasecmp(FIELD_DB,"MSSQLServer")≠0).

Note that, the vulnerability condition for this vulnerability states that the length of the DB field needs to be larger than 64 bytes. Thus, the last two conditions in the VPRP are redundant and the final protocol-level signature would be: `(FIELD_CMD == 4) && length(FIELD_DB) > 64` . According to the ShieldGen authors, who had access to the source code, this signature would be optimal.

Gdi-wmf. Figure 5 shows on the left the field structure for the exploit file and on the right the VPRP. The original protocol-level path-predicate contained the 4 aligned nodes on the left of the graph, while the exploration discovers one new path leading to the vulnerability point that introduces the node on the right. The graph shows that the program checks whether the `Version` field is 0x300 (Windows 3.0) or 0x100 (Windows 1.0). Such constraint is unlikely to be detected by probing approaches, since they usually sample only a few values. In fact, in ShieldGen they analyze a different vulnerability in the same library but run across the same constraint. The authors acknowledge that they miss the second condition of the disjunction. Thus, an attacker could easily avoid detection by changing the value of the Version field. Since we have no access to the source we cannot verify if our VPRP is optimal, though we believe it to be.

Other experiments. Due to space constraints we refer the reader to our extended version [36] for details on the Atphttpd, GHttpd and DCOM RPC examples. For the Atphttpd and GHttpd vulnerabilities, where we have access to the source code, the extended version contains the optimal signatures that we manually extracted for the vulnerability. The results show that Elcano's VPRPs exactly match or are very close to the optimal ones that we manually extracted from the source code.

6 Conclusion

In this paper we propose protocol-level constraint-guided exploration, a novel approach to automatically generate high coverage, yet compact, vulnerability point reachability

predicates, with application to signature generation, exploit generation and patch verification. Our experimental results demonstrate that our approach is effective, generates small vulnerability point reachability predicates with high coverage (optimal or close to optimal in cases), and offers significant improvements over previous approaches.

Acknowledgements

We would like to thank James Newsome and Prateek Saxena for many helpful discussions on signature generation. We also thank Stephen McCamant and the anonymous reviewers for their insightful comments on this document.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

1. Symantec: Internet security threat report (2008), http://www.symantec.com/business/theme.jsp?themeid=threatreport
2. Kreibich, C., Crowcroft, J.: Honeycomb - creating intrusion detection signatures using honeypots. In: Workshop on Hot Topics in Networks, Boston, MA (2003)
3. Kim, H.A., Karp, B.: Autograph: Toward automated, distributed worm signature detection. In: USENIX Security Symposium, San Diego, CA (2004)
4. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Symposium on Operating System Design and Implementation, San Francisco, CA (2004)
5. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: IEEE Symposium on Security and Privacy, Oakland, CA (2005)
6. Yegneswaran, V., Giffin, J.T., Barford, P., Jha, S.: An architecture for generating semantics-aware signatures. In: USENIX Security Symposium, Baltimore, MD (2005)
7. Li, Z., Sanghi, M., Chen, Y., Kao, M.Y., Chavez, B.: Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In: IEEE Symposium on Security and Privacy, Oakland, CA (2006)
8. Liang, Z., Sekar, R.: Fast and automated generation of attack signatures: A basis for building self-protecting servers. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2005)
9. Liang, Z., Sekar, R.: Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In: Annual Computer Security Applications Conference, Tucson, AZ (2005)
10. Wang, X., Li, Z., Xu, J., Reiter, M.K., Kil, C., Choi, J.Y.: Packet vaccine: Black-box exploit detection and signature generation. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2006)
11. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: Symposium on Operating Systems Principles, Brighton, United Kingdom (2005)

12. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: IEEE Symposium on Security and Privacy, Oakland, CA (2006)
13. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: International Symposium on Software Testing and Analysis, Chicago, IL (2009)
14. Vigna, G., Robertson, W., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. In: ACM Conference on Computer and Communications Security, Washington, DC (2004)
15. Rubin, S., Jha, S., Miller, B.P.: Automatic generation and analysis of nids attacks. In: Annual Computer Security Applications Conference, Tucson, AZ (2004)
16. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signatures using weakest pre-conditions. In: Computer Security Foundations Symposium, Venice, Italy (2007)
17. Costa, M., Castro, M., Zhou, L., Zhang, L., Peinado, M.: Bouncer: Securing software by blocking bad input. In: Symposium on Operating Systems Principles, Bretton Woods, NH (2007)
18. Cui, W., Peinado, M., Wang, H.J., Locasto, M.: Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In: IEEE Symposium on Security and Privacy, Oakland, CA (2007)
19. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: IEEE Symposium on Security and Privacy, Oakland, CA (2008)
20. A dumb patch? http://blogs.technet.com/msrc/archive/2005/10/31/413402.aspx
21. Common vulnerabilities and exposures (cve), http://cve.mitre.org/cve/
22. Wireshark, http://www.wireshark.org
23. Pang, R., Paxson, V., Sommer, R., Peterson, L.: Binpac: A yacc for writing application protocol parsers. In: Internet Measurement Conference, Rio de Janeiro, Brazil (2006)
24. Borisov, N., Brumley, D., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: A generic application-level protocol analyzer and its language. In: Network and Distributed System Security Symposium, San Diego, CA (2007)
25. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2007)
26. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: Automatic reverse engineering of input formats. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2008)
27. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
28. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
29. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed System Security Symposium, San Diego, CA (2005)
30. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D., Engler, D.R.: Exe: Automatically generating inputs of death. In: ACM Conference on Computer and Communications Security, Alexandria, VA (2006)
31. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL (2005)

32. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
33. Vine, http://bitblaze.cs.berkeley.edu/vine.html
34. Caballero, J., McCamant, S., Barth, A., Song, D.: Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley (2009)
35. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: International Conference on Information Systems Security, Hyderabad, India (2008); Keynote invited paper
36. Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration (extended version), http://www.ece.cmu.edu/~juanca/papers/fieldsig_extended.pdf
37. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ (2008)
38. Boonstoppel, P., Cadar, C., Engler, D.: Rwset: Attacking path explosion in constraint-based test generation. In: International Symposium on Software Testing and Analysis, Seattle, WA (2008)
39. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8) (1975)

Automated Behavioral Fingerprinting

Jérôme François[1,2], Humberto Abdelnur[1], Radu State[3], and Olivier Festor[1]

[1] MADYNES - INRIA Nancy-Grand Est, France
{firstname.lastname}@loria.fr
[2] CNRS, Nancy-Université, France
[3] University of Luxembourg, Luxembourg
radu.state@uni.lu

Abstract. This paper addresses the fingerprinting of devices that speak a common, yet unknown to the fingerprinting engine, protocol. We consider a behavioral approach, where the fingerprinting of an unknown protocol is based on detecting and exploiting differences in the observed behavior from two or more devices. Our approach assumes zero knowledge about the syntax and state machine underlying the protocol. The main contribution of this paper consists in a two phased method. The first phase identifies the different message types using an unsupervised support vector clustering algorithm. The second phase is leveraging recent advances in tree support kernel in order to learn and differentiate different implementations of that protocol. The key idea is to represent behavior in terms of trees and learn the distinctive subtrees that are specific to one particular device. Our solution is passive and does not assume active and stimulus triggered behavior templates. We instantiate our solution to the particular case of a VoIP specific protocol (SIP) and validate it using extensive data sets collected on a large size VoIP testbed.

1 Introduction

Over the past few years, there has been an increased effort in the research community towards the automated analysis and reverse engineering of network protocols. The driving forces are multiple and range from practical needs to analyze network traffic generated by malware where the most notorious case is the Storm bot and up to the development of open source implementation for poorly documented protocols, as it was the case of the SMB protocol [1] for example. A related problem is the automated and passive fingerprinting of devices using an unknown protocol. While some research efforts in this direction have been recently made in [2] in order to learn the syntax and grammar that generated the protocol messages, to our knowledge, none until now has addressed the automated learning of the specific behavior of a protocol in order to fingerprint a device or a protocol stack. [3] and [4] are close and complementary works as they aim to learn an unknown protocol to automatically respond to requests.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 182–201, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The research challenges that we face are related to learning the relevant protocol operations/primitives and modeling the protocol message sequences such that automated learning is possible. If packet captures from an unknown protocol are given, we aim first to automatically discover the unknown types of messages. We assume furthermore that no learning set with labeled protocol messages exists, that no encryption is used and that no reverse engineering of the application using such a protocol is possible. We also assume that the number of different message types is a-priori unknown.

Reverse engineering of network protocols is clearly related to the security problem since understanding the protocols is a necessary step for detecting protocol misuse. In addition, fingerprinting is a useful task in the security domain. For an attacker, fingeprinting is a prior work for performing efficient attacks. For a network administrator, it is a tool for security assessment and testing.

The two contributions presented in this paper are:

- the automated analysis of protocols with respect to the types of messages exchanged based on unsupervised learning methods. For instance, if we consider a stripped-off ICMP version, an ICMP echo request message has to be answered by an ICMP echo reply message; assuming that a collection of captured messages is available, our approach should automatically discover that several types of messages exist (ICMP echo request and ICMP echo reply). Thus, our technique can be used as an essential preprocessing phase to the automated learning of the protocol related state machine;
- the learning of the device/stack specific behavior that results from reconstructing/reverse engineering a state machine for a device under test. In the previous example of ICMP for instance, it may be possible to detect a specific device by peculiar behavior related features. Although ICMP is a simple protocol, a large and comprehensive research work [5] (using mostly manual and tedious tests) showed that reliable fingerprinting is possible. We address a new and automated research direction that leverages support vector machines and tree kernels for learning structural behavior features of the underlying protocol.

The paper is structured as follows: related work is analyzed in the next section; the SIP protocol that we use as the first application case is described in the third section; the different metrics to classify the messages are presented in section 4. Message types identification methods and results are described in section 5. Section 6 focuses on the behavioral fingerprinting. Finally, the last section concludes the paper and sketches future research directions.

2 Related Works

Automatically recognizing the different messages of a protocol without prior knowledge is a challenging task. This is one part of the reverse protocol engineering goals which also aims to clearly infer the syntax and grammar of messages *i.e.*, the different fields. Historically, the first technique available was hand-based

analysis of dump files provided by packet sniffing software like tcpdump [6]. Obviously, this technique is tedious, limited and very time consuming. Therefore, new methods appeared. The Protocol Informatics project [7] proposes a solution which uses well known bioinformatics algorithms and techniques based on sequence alignment. Given a set of messages protocols, the program tries to determine both constant and variable fields. Several approaches consider the context semantics *i.e.*, the target computer behavior itself: [2] looks for extracted bytes in the message to rebuild the different fields; [8] is based on the execution trace *i.e.*, system calls; [9] proposes a dynamic binary analysis to identify separators and keywords; [10] introduces a semi-supervised learning method to determine the message fields. Closer to our goal, an approach to cluster the messages captured on the network by types before trying to infer their format is proposed in [11]. To achieve this clustering, the authors propose to tokenize each message *i.e.*, to find the different fields by considering that each binary bytes is a binary token and that each text sequence between two binary bytes is a field. The technique proposed in [12] is also based on identifying the different fields thanks to a delimiter. This is done by instrumenting the protocol application by studying how the program parses the messages. [13] focuses more on the state machine construction of multiple flows protocol. Application dialog replay is a very close domain since its goal is to construct a valid replay dialog by identifying the contents which need to be modified thanks to sequence alignment techniques [3] or by building a model from application inputs and outputs [14]. ScriptGen [4] is another approach which is able to construct the partial state-machine of a protocol based on network traces in order to automatically generate responses to attacker requests sent to a honeypot. Network and service fingerprinting is a common task is security assessment, penetration testing and intrusion detection. The key assumption is that subtle differences due to development choices and/or incomplete specification can trace back the specific device/protocol stack [15]. There are excellent tools that implement such schemes: p0f [16] uses TCP/IP fields to passively identify the signature of a TCP/IP stack, while nmap [17] does actively follow a stimulus-response test in order to detect the operating system and service versioning of a remote device. [18] aims to construct automatically the fingerprints by active probing. The research community has addressed the fingerprinting of SIP devices [19,20] by automatically constructing message specific fingerprints. In [21] and [22], the goal is a little bit different because the authors aim to correctly identify the flow types *i.e.*, the protocols used. In a previous contribution [23], we have addressed a syntax driven fingerprinting, where parse trees of captured messages were used to learn distinctive features capable to perform fingerprinting. In that study, we assumed that BNF [24] specifications are available and that individual messages can be used to infer vendor/stack specific implementation characteristics. This is different from the current approach where no a-priori knowledge of the syntax is assumed. Secondly, we did not consider until now the behavioral aspects for the fingerprinting task. In this paper we do consider the latter and we leverage differences in induced state machines in order to perform fingerprinting.

3 Session Initiation Protocol (SIP)

SIP [25] is the de-facto signalisation protocol for the management of VoIP communications. Its main features are related to the establishment, tear-down and negotiation of VoIP sessions and it comprises a rich set of protocol primitives and options as well as a complex underlying state machine. We consider SIP to illustrate our approach for several reasons. Firstly, the number of operations is relatively important (more than 10). Secondly, due to its design, a clear delimitation of transport level information and network level information does not exist, thus making the automated analysis difficult. Thirdly, the distribution of individual message types is quite unbalanced: some message types appear very rarely such that small sided and under-represented classes have to be dealt with. Although a complete overview of SIP is beyond the scope of this paper, a short introduction is given below. SIP messages are divided into two categories: requests and responses. Each request begins with one of the following keywords: `REGISTER, OPTIONS, INVITE,UPDATE, CANCEL, ACK, BYE, SUBSCRIBE, NOTIFY, PRACK, PUBLISH, INFO, REFER, MESSAGE`. The SIP responses begin with a numerical code of 3 digits divided into 6 classes identified by the first digit.

A SIP transaction is illustrated in the figure 1. It is an usual example when *user1@domain1.com* wants to call *user2@domain2.com*. So *user1* initiates the connection by sending an *INVITE* request. First, the callee *user2* informs that it receives the request and will try to achieve its by the *Trying* message. The *Ringing* message means that the user is alerted of the incoming call. When *user2* decides to accept the call, the *OK* response is sent. The caller acknowledges this one and the media session over RTP (Realtime Transport Protocol) [26] is established. Finally, *user2* hangs the phone, a *BYE* message is sent and the other party send an *OK* response to accept the ending of the session. Obviously, many details are omitted like the negotiation of parameters.

We have built a dataset of 1580 SIP messages, generated using several phones coming from different manufacturers. In our traces we minded 27 different kinds

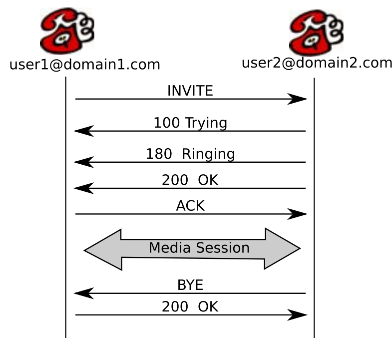


Fig. 1. SIP transaction example

Fig. 2. SIP messages distribution

of message which are the most important and used in VoIP networks. Their empirical distribution is illustrated in figure 2.

Regarding SIP as a target protocol, we would like to be able to automatically identify the 27 types of messages and automatically build tools that can identify abnormal sequences which are not compliant with the protocol, but can be linked to a specific SIP stack and/or vendor. We had to use a known protocol (SIP) in order to be able to assess the accuracy and precision of our approach but it can be generalized towards any protocol. One of the key components is the differentiation among several method types with appropriate metrics.

4 Metrics and Distances for Protocol Messages

Character distribution. The character distribution d of a message is based on the statistical distribution of its characters. Intuitively, if two messages m_a and m_b are of the same type, their distributions should be similar. The natural and logical measure is the symmetric relative entropy or Kullback-Leibler divergence:

$$char_dist(m_a, m_b) = \sum_i d(m_a)_i log(\frac{d(m_a)_i}{d(m_b)_i}) + \sum_i d(m_b)_i log(\frac{d(m_b)_i}{d(m_a)_i}) \quad (1)$$

where i represents all possible characters.

Relative character distribution. An alternative character distribution can be constructed to capture the relative character distribution of the message m: rel_char_dist. It is based on the distribution of characters but can deal with simple enciphering schemes like for example the per character XOR encryption. It uses an ordered distribution [27] of the character frequencies. Thanks to this metric we have:

$$rel_char_dist(m \oplus k) = rel_char_dist(m) \forall\ key\ k \quad (2)$$

Character position. A third metric that can be associated to each message is the character position. Basically, each message of a protocol has different fields and each message of the same type usually has common field filled with similar content. Therefore, the character at a certain position is often the same for a certain type of message. This metric $char_pos(m)$ determines the average position of each character of the message m:

$$char_pos(m)(c) = \frac{\sum_{i=1}^{i=k} pos(a_i)}{k} \quad (3)$$

where i is the index of the character c with k occurrences in the message and $pos()$ the function returning the position of the index of a given character.

Weighted character position. Most protocol messages are formed by a header containing the type of the message followed by options, arguments and an additional payload. This comes from good and established protocol design patterns. The $weighted_char_pos(m)$ balances more the first characters:

$$\forall\ character\ c\ occurring\ k\ times,\ p_2(m)(c) = \frac{\sum_{i=1}^{i=k} pos(a_i)^{-1}}{k} \quad (4)$$

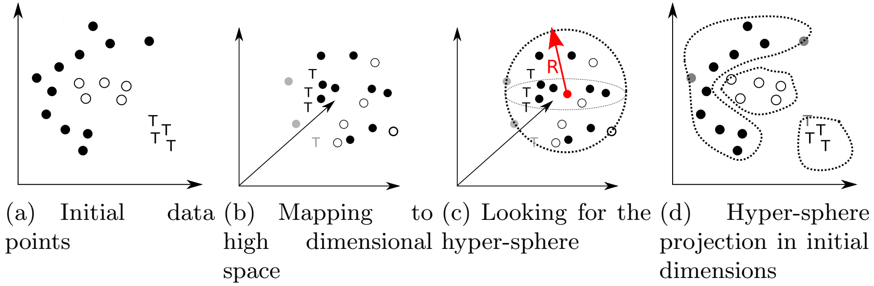
The key assumption is that messages of the same types should start with similar headers even if the message contents are totally different.

5 Automated Protocol Clustering

Using the previously defined metrics, we derive an unsupervised clustering method that combines two clustering methods in order to determine the number of different messages types. The first technique is a new method relying on unsupervised support vector clustering [28]. The second method is based on the well known agglomerative nearest neighbor method [29]. This last technique considers each data point as an individual cluster. The two clusters with the smallest inter-distance are merged into one. Then, this step is repeated until the smallest inter-distance is higher than a threshold t.

5.1 Support Vector Clustering

The support vector clustering (SVC) technique has been introduced in [28] and leverages machine learning paradigms based on support vector machines (SVM) [30] techniques. Such techniques show good accuracy with a limited overhead in different domains [31]. The initial data points 3(a) are mapped from the input space to a high dimensional space using a non linear transformation 3(b). The goal is to find the smallest sphere which contains all the points in the high dimensional space 3(c). This sphere is mapped back to the original input space and forms a set of countours which are considered as the cluster boundaries 3(d). The final step determines the cluster of each point by checking which boundaries contain it.



(a) Initial data points (b) Mapping to high dimensional space (c) Looking for the hyper-sphere (d) Hyper-sphere projection in initial dimensions

Fig. 3. SVC example

Consider Φ, a nonlinear transformation and $\{x_i\}$ the set of N points in the original d-dimensional input space. The training phase consists of finding the smallest hyper-sphere containing all the transformed points *i.e.*, $\{\Phi(x_i)\}$ which is characterized by its radius R and its center a. Therefore we have:

$$\|\Phi(x_i) - a\|^2 \leq R^2 \; \forall i \tag{5}$$

The original problem is casted into the Lagrangian form by introducing the lagrangian multipliers (β_i and μ_i) and the penalty term ($C \sum_i \xi_i$):

$$L = R^2 - \sum_i (R^2 + \xi_i - \|\Phi(x_i) - a\|^2)\beta_i - \sum_i \xi_i \mu_i + C \sum_i \xi_i \tag{6}$$

In fact, the ξ terms are slack variables allowing some classification errors. Then, the problem is turned into its Wolfe dual form and the variables a and R are eliminated due to Lagrangian constraints.:

$$W = \sum_i \Phi(x_i)^2 \beta_i - \sum_{i,j} \beta_i \beta_j K(x_i, x_j) \tag{7}$$

where $K(x_i, x_j)$ is typically defined by a Gaussian Kernel:

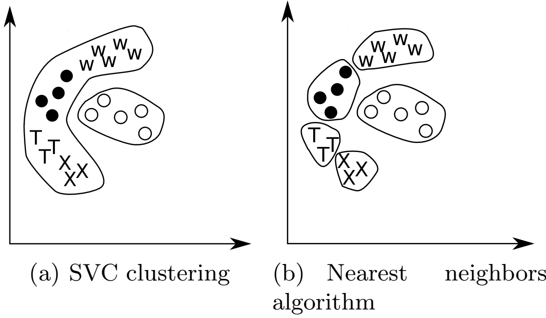
$$K(x_i, x_j) = e^{-q\|x_i - x_j\|^2} \tag{8}$$

where q is another parameter named `Gaussian` width.

Next, a labeling step has to determine the points that belong to the same clusters by a geometric approach. In fact, two points are considered of the same clusters if all the points on the segment between them in the original space are in the hypersphere in the high dimensional feature space.

5.2 Global Method

Even if SVC enables the discovery of intertwined clusters, the accuracy can be limited when a single shape comprises different clusters. The figure 4 shows such a case, where in figure 4(a), the SVC method is able to isolate two large clusters but none the single ones which composes the largest one. These constructed



(a) SVC clustering (b) Nearest neighbors algorithm

Fig. 4. Global Method

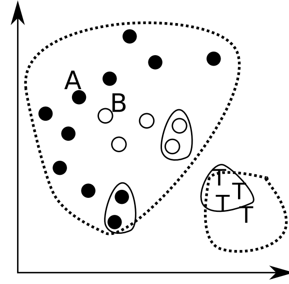


Fig. 5. Limitation of nearest neighbors clustering

clusters can be furthermore split by an additional nearest neighbors technique for each of them. Hence, this second step is necessary. Obviously, it depends also on the data to classify and our experiments in the next sections show the benefits of the combination of these two methods. Furthermore, several multi-pass clustering methods exist and are introduced in [32]. Complex clusters boundaries are discovered by the SVC technique. By applying the nearest neighbors technique, the result shown in figure 4(b) can be obtained. However, applying only the nearest neighbors technique will entail a bad classification as illustrated in figure 5. Therefore, we propose a global method which consists in two steps:

– a first clustering using SVC
– a second cluster splitting using nearest neighbors technique

5.3 Evaluation Metrics

We consider several metrics in order to assess the quality of the clustering method Consider n messages to be classified, $m_1 \dots m_n$, divided into r types and k clusters found: $c_1 \dots c_k$ with $k \leq n$. At the end of the classification, a label is assigned to each cluster which is the predominant type of the messages within. However, only one cluster per type, the largest one, is allowed. If $c(m_i)$ represents the cluster containing m_i then $t(m_i)$ is the real type of the message m_i and $t(c_i)$ is the type assigned to the cluster (c_i).

The first metric is the classification rate cr and represents the ratio of messages which are classified in the right clusters:

$$cr = \frac{\sum_{i|t(m_i)=t(c(m_i))} 1}{n} \tag{9}$$

The second metric is the proportion of different message types which were discovered:

$$cf = \frac{r}{k} \tag{10}$$

The latter is a rather important metric because performing a good classification rate can be easy by discovering the main types and ignoring unusual ones. In our

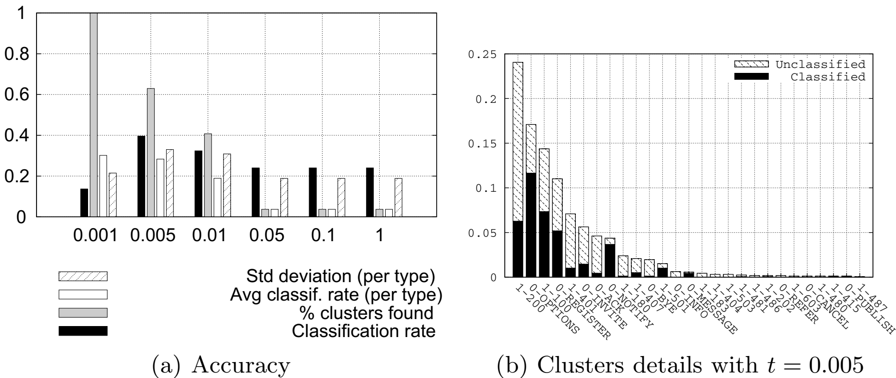
dataset for instance, having a classification rate close to 100% can be obtained without discovering small clusters like 603, 480, 486, 487 as shown in figure 2. Some of these classes have only one recognized message. The classification rate can also be computed for each type y:

$$cr_{type}(y) = \frac{\sum_{i|t(m_i)=y} x_i}{\sum_{i|t(m_i)=y} 1} \; where \; x_i = 1 \; if \; t(m_i) = t(c(m_i)) \; else \; 0 \tag{11}$$

In practice we will consider the average value and the standard deviation by computing this metric for all possible kinds. Therefore, the classification accuracy has to be discussed regarding these different metrics. Because, several figures relate to the accuracy, a common key will be used and will be displayed only in figure 6(a). We analyze the composition of the different clusters with two metrics. The first one is the percentage of good classified messages which are contained in the considered cluster type. The second one is the percentage of messages of this type which are not present in the cluster.

5.4 Nearest Neighbors Technique Results

We consider the first metric to be the relative character distribution. The results are presented on figure 6 where the parameter t varies. This parameter is the maximal distance authorized between two points. The best tradeoff between the classification rate and the number of clusters found is obtained for $t = 0.005$ on the figure 6(a). In this case, about 40% of messages are classified correctly and 60% of the types are found. This shows that for 40% of types the average classification rate is 0% and the standard deviation of the classification rate per type is relatively high. The third bar represents the average classification rate per type which is close to the global classification. The composition of the clusters is interesting since they are similar (see figure 6(b)). In fact, there is no type which is totally well classified. So each cluster contains a relatively high proportion of misclassified messages. t is the main parameter of the nearest neighbors algorithm and



(a) Accuracy

(b) Clusters details with $t = 0.005$

Fig. 6. Relative character distribution results

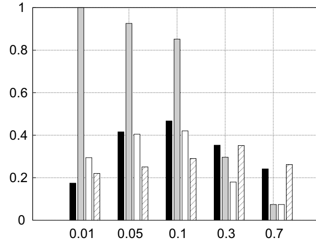
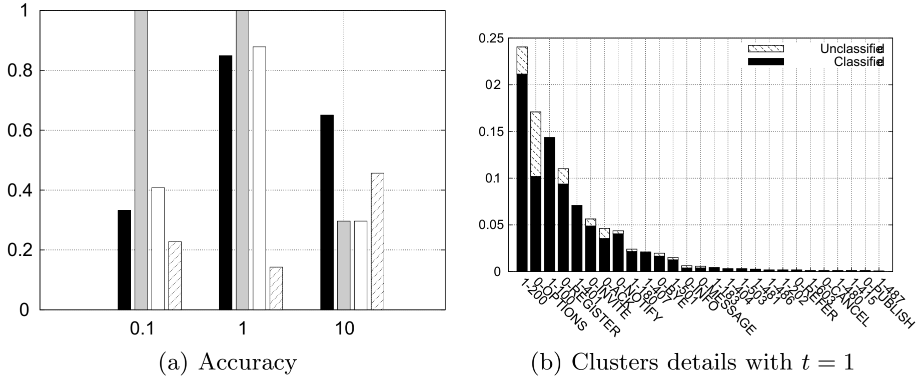


Fig. 7. Smoothing character distribution results - Accuracy

does have a high impact on the accuracy (see figure 6(a)). When t increases, less clusters are found because the maximum distance between two points of a cluster is increased and so the cluster sizes are larger. Hence, when t increases, the clusters are merged. The classification rate variation is less obvious: when t increases, it begins by increasing and followed by a decrease. The reason is correlated to the number of clusters. With a small t, the cluster sizes are small too leading messages of the same types to be split into multiple clusters. This entails a bad classification rate because only the biggest one is considered for each kind. When t increases, the clusters are merged, especially many clusters of the same kind. Then, clusters of different types can be grouped into one and in this case all messages of one type are missclassified in the new cluster which decreases the classification rate.

The next experiment is based on the character distribution which captures the information in the characters. To limit the effect of the zero values, the results using the smoothing distribution is presented on the figure 7. We checked with other experiments that the smoothing technique has a low impact on the classification rate but allows to discover easily more kinds of message. Comparing the relative character distribution results, they are not significantly improved except for the number of clusters found: about 90% with a classification rate of about 40% ($t = 0.05$). The number of found clusters is better with the same classification rate. This is confirmed by the increase of the average classification rate per type. This means that some small clusters are found too. Moreover, the associated standard deviation is reduced for the same reason.

The character position metric is accounting for that the first characters in a message are most probably relevant for the message identification. For instance, the INVITE message has two "I" in the first 7 bytes, and thus a good characteristic of this message is that this letter is more present at the beginning. However, an INVITE message contains basically a long payload formed with uri, parameters and so on. It may also contain "I" because its length is much higher than the INVITE keyword. The weighted character position metric gives more importance to first characters. The results plotted in figure 8(a) are very good as it is possible to find all the different kinds of message with a global and per type classification rate close to 85%. The details of the classification are illustrated in figure 8(b). In fact, the misclassified messages are shared out within several



(a) Accuracy

(b) Clusters details with $t = 1$

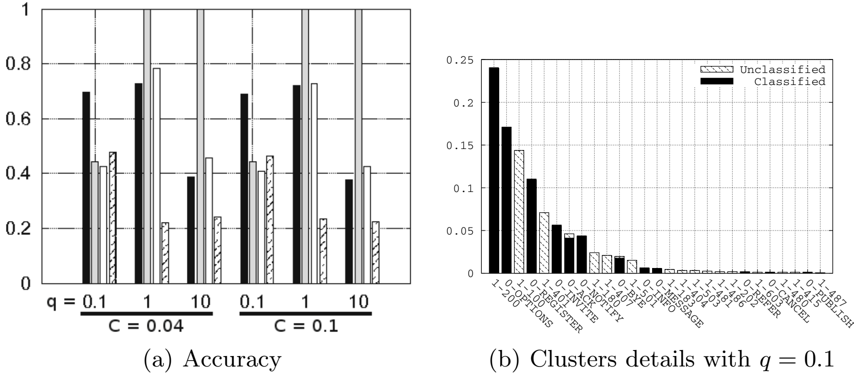
Fig. 8. Weighted characters position results

clusters which entails a small standard deviation for the classification rate per type on figure 8(b). So, increasing the accuracy has not to focus on single or few types of messages only.

5.5 SVC Technique Results

We applied the SVC method with different values for the Gaussian width q and the penalty factor C. The weighted character position metric is used because it is the best to differentiate the messages. As it is shown in the figure 9(a), the best possible accuracy is 0.73 for the classified messages with all types of messages found. This result is good but slightly lower than the nearest neighbors technique on figure 8 (85% of good classification). This is mainly due to a poor discovery of the smallest clusters because the standard deviation of the specific classification rate per type is higher. When the Gaussian width q increases between 0.1 and 1, the difference between the packets is emphasized in the high dimensional feature space. Hence, the messages clusters are more split and the accuracy is improved. However, when q is too high, the number of clusters continues to increase with redundant types. The number of found clusters is then still good but due to many redundant cluster types, the classification rate drops.

The cluster composition is interesting. In this case, the best accuracy provides clusters similar to the previous obtained in the figure 8 where all kinds of clusters are represented with a little proportion of missclassified messages inside each one. However, if we consider the case of $C = 0.04$ and $q = 0.1$ in figure 9(b) with a lower accuracy, the clusters are totally different, all types are not found but for most of them, all messages are totally discovered. It means that these clusters represented by black bars contain also messages of other types because the latter ones are not classified in their own clusters -represented by stripes bar (unclassified). We apply the nearest neighbors technique on each cluster to split them. By looking for the best t value, we found $t = 1.1$ which allows to classify 91% of the messages and to discover 96% of the types of the message.



(a) Accuracy

(b) Clusters details with $q = 0.1$

Fig. 9. Weighted characters position SVC results

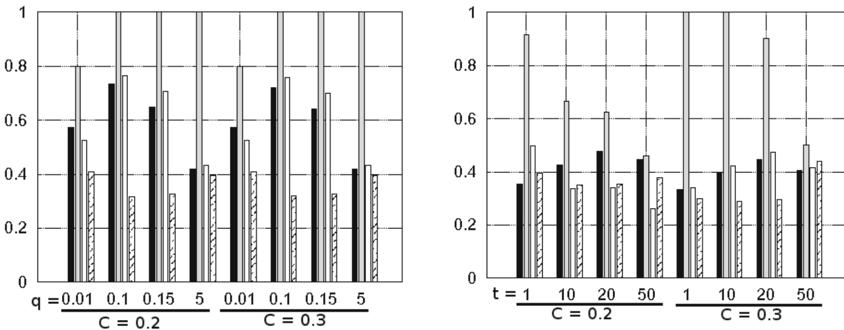


Fig. 10. SVC - SMTP dataset

Fig. 11. Nearest neighbors clustering

5.6 Other Protocols

We have applied also our method to other protocols. We considered only the weighted character position metric because it provided the best results in the previous section. Two well known protocols were tested: SMTP [33] (150 packets and 10 types) and IMAP [34] (289 packets and 24 types). To ease the comparison of results, a classical standardization of the data is done. When the nearest neighbors technique is applied, the classification rates are similar for these protocols -as shown in figure 11 - and less than 50% of the messages are well identified. The number of clusters found is better for SMTP. Moreover, using standardized data helps to choose the parameter t in order to obtain the best classification rate. Then $t = 20$ seems to be a good value to apply the nearest neighbors method with the standardized weighted character position metric.

The SVC method instantiated with the IMAP does not improve the clustering accuracy since in the best case, only 36% of the messages are well classified. Hence, doing the second step with nearest neighbors technique is necessary and allows to obtain 49% of good classification. This is slightly better than the nearest neighbors

technique as it was 47% on the figure 11. Obviously, this difference is very low but the number of different types found increases from 62% to 96% with SVC. Therefore, even if the combined method doesn't improve the classification rate, it is able to keep the classification rate stable and at the same time discovering more message types. The figure 10 shows the accuracy of SVC for the SMTP traffic. Since the nearest neighbors technique was able to find most of the types in figure 11, SVC can also find them. Moreover, the classification rate is greatly improved: 72% of messages are correctly identified and 80% of kinds are found. By applying the nearest neighbors technique on the obtained clusters, the results are very close because only one additional type is identified with one packet. Hence, the number of discovered types is 90%. The standard deviation of the classification rate per type is quite high (0.39) principally due to one type totally ignored in both cases.

5.7 Semi Automated Parameters Identification

The assessing of the classification results for a known protocol is easy. The same is much more difficult with unknown because no reference exists. The first conclusion of our study is that the standardized weighted character position metric is the most suitable. For SVC technique, there are two parameters: C and q. In our experiments C has not a great impact as it is highlighted in figure 10. This value is constrained by the SVC technique itself and we have $\beta_init_i < C < 1$ where β_init_i are initial values of β_i in the Wolfe dual form. These values can be randomly selected since their impacts is only limited to the computation time. Their sum has to be one and we can choose $\beta_init_i = \beta = 1/\#number_of_data_points$. In our case, the number of data points is the number of packets in the dataset. We can consider a minimal dataset size of 100 and so $C = 0.2$ or $C = 0.3$ are suitable. The parameter q is the Gaussian width and has a real impact on the classification. The aim of the SVC clustering is to obtain large clusters regrouping several types within in. Then, the width of a classification instance is the width of the largest cluster. The width of a cluster is the maximal distance (depending on the metric used) between two points within it. For small values of q, all data points are regrouped into the same cluster and when it increases, they are split. Therefore, when the big clusters will be divided into several ones, if the width is about the same, there will be still a big cluster remaining. If the width is different, the cluster has been really divided into several important ones. In the same time, the number of clusters has to increase more during this step. Considering p values of q: $q_1, q_2, \ldots, q_p$, the associated number of clusters $(\#c(q_1), \ldots, \#c(q_p))$ and the classification width $(w(q_1), \ldots, w(q_p))$. To observe the evolutions of these values, we define the following metrics where the division are required to have a value between 0 and 1:

$$evol_c(q_i) = \frac{|\#c(q_i) - \#c(q_{i-1})|}{max_i(\#c(q_i))} \; if \; i > 1 \; else \; 0 \tag{12}$$

$$evol_w(q_i) = \frac{|w(q_i) - w(q_{i-1})|}{max_i(w(q_i))} \; if \; i > 1 \; else \; 0 \tag{13}$$

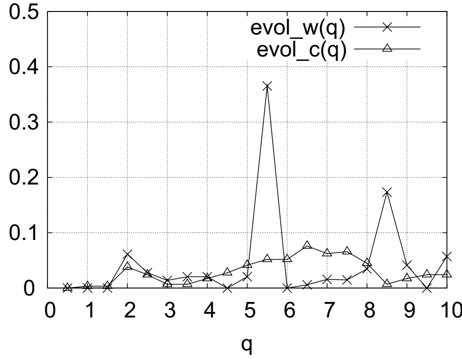


Fig. 12. SVC Gaussian width q selection (IMAP)

These are plotted in the figure 12 for the IMAP protocol. Several peaks of $evol_w(q)$ exist, but the last one for $q = 8.5$ can be easily discarded because the number of clusters decreases in the same time. The second one is interesting since it represents the value selected by hand ($q = 5.5$) and the clusters number increases in the same time. This semi automated technique is able to find a good parameter. Finally, the first peak ($q = 2$) is not so high but it concerns simultaneously both metrics. By testing this value, the classification rate is slightly improved by reaching 50%. With others protocols, this approach is able to identify the same optimal parameters which were found by manual testing. Identifying optimal parameters for the nearest neighbors technique can be based on known techniques like [35].

To conclude, combination of weighted normalized position metric and SVC technique is often able to improve the recognition of the messages types. By doing a second phase based on the nearest neighbors technique, the results are always improved.

6 Behavioral Fingerprinting

In order to best describe our approach, we will use the example illustrated in figure 13. The messages exchanged between two parties are captured first. Secondly, each message can be mapped to the corresponding message type. This is done using the clustering mechanism described previously. Once each message is mapped to its cluster, a session of captured messages can be represented as a sequence of clusters. A session is composed of the messages exchanged between two entities without a relative long inactivity period. in fact, TCP based protocol sessions are easily distinguishable

The original sequence of messages can be mapped to the sequence of clusters:

$$\{m_1(A \rightarrow B), m_2(B \rightarrow A), m_3(A \rightarrow B), m_4(A \rightarrow B)\}$$
$$\equiv \{c(m_1)(A \rightarrow B), c(m_2)(B \rightarrow A), c(m_3)(A \rightarrow B), c(m_4)(A \rightarrow B)\}$$
$$\equiv \{c_1(A \rightarrow B), c_2(B \rightarrow A), c_2(A \rightarrow B), c_3(A \rightarrow B)\}$$

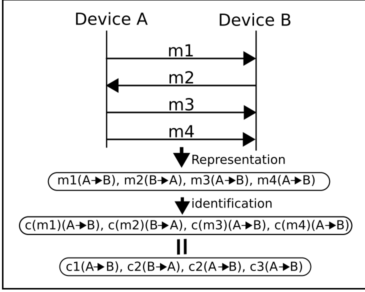


Fig. 13. Session example

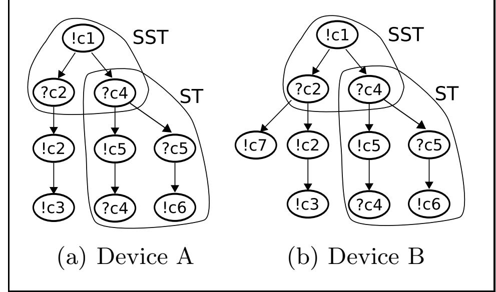


Fig. 14. Kernel trees

$\equiv \{!c_1, ?c_2, !c_2, !c_3\}_{A \to B}$

We use the notation ?x for a message x that is received and the notation !x to model that a message of type x is emitted by a device.

A second capture might consist in the sequence of messages (figure 13):

$$\{m_5(A \to B), m_6(B \to A), m_7(A \to B), m_8(B \to A)\}$$
$$\equiv \{c(m_5)(A \to B), c(m_6)(B \to A), c(m_7)(A \to B), c(m_8)(B \to A)\}$$
$$\equiv \{c_1(A \to B), c_4(B \to A), c_5(A \to B), c_4(B \to A)\}$$
$$\equiv \{!c_1, ?c_4, !c_5, ?c_4\}_{A \to B}$$

In the same manner, the final capture consists in the sequence:

$$\{!c_1, ?c_4, ?c_5, !c_6\}_{A \to B}$$

Then a tree based representation can summarize them as follows: a simple algorithm adds incrementally a sequence to the tree. This is done by checking the longest prefix of the chain that is also a path sourced (in the root) in the tree. Starting with the last node in this path, the remaining suffix is added as a chain to the tree.

We construct for each device, a device specific trees - see figure 14. Each node in the tree corresponds to a message type. An edge in the tree links two nodes if the corresponding message types have been observed to succeed in the traces Although known to be NP complete (see [36],[37] and [38] for good overviews on this topic), the existing heuristics for doing it are based on building tree representations for the underlying finite state machine. In our approach we don't prune the tree and although the final tree representation is dependent on the order in which we constructed the tree, we argue that the resulting substrees are good discriminative features. We follow a supervised training method, where protocol trees are labeled with the identity of their class. The identity of the class is assumed to be known. For instance, the figure 14(a) shows that based on traces, a node can start by sending an INVITE message ($!c1$) and receiving afterwards 180 ($?c2$) or 407 ($?c4$) typed messages. In SIP, a 180 typed message is used to learn that the call in in progress, while 400 messages are related to authentication requests.

The core idea behind behavioral fingerprinting consists in identifying subtrees in the underlying tree representations that can uniquely differentiate between two observed behaviors. We developed a classification method based on trees kernels in order to take into account the peculiar nature of the input space. Tree kernels for support vector machines have been recently introduced in [39], [40], [41] and do allow to use substructures of the original sets as features. These substructures are natural candidates to evaluate the similitude and differentiate among tree-like structures. We have considered two kernel types introduced in [42],[43] and [41]: the subtree (ST) kernel and the subset tree kernel (SST). Simply stated a subtree (ST) of a node is just the complete subtree rooted in that node. A subset tree corresponds to a cut in the tree - a subtree rooted in that node that does not include the original leaves of the tree. For instance, the figure 14 highlights some examples of similar SST and ST for two trees. Figure 14(a) represents a very simple tree corresponding to a Linksys SIP Phone. In the context of behavioral fingerprinting, a device specific protocol tree can be mapped to a set of ST and SST features by extracting all underlying SSTs and STs. Two protocol trees generated by two different devices (figure 14) can now be compared by decomposing each tree in its SSTs and STs followed by a pair-wise comparison of the resulted SSTs and STs. This can be done using tree kernels as proposed in [41]. The idea behind tree kernels is to count the number of similar SSTs in both features sets and/or check the exact matching of underlying STs. The interested reader is referred to [41] for more completeness and fast implementation techniques.

For our purposes, similar substructures correspond to similar behavior in terms of exchanged messages and represent thus a good measure of how much two devices are similar with respect to their behavior. We collected traces from a real VoIP testbed using more than 40 different SIP phones and SIP proxies. In the learning phase, we trained the support vector machines using a modified version of the svm-light -TK [44] developed by Alessandro Moschitti. Our dataset consisted in complete SIP traces obtained during a one day capture from a major VoIP provider. The capture file (8 GB) contained only the signaling SIP related data. Using the user-agent banner, we could identify 40 different end devices. We have also observed traffic coming from user-agents that were not identifiable. This latter is due probably to some topology hiding performed by home routers or session border controllers. For each device/user-agent we constructed the underlying tree representations using a maximum of 300 SIP dialogs. Therefore, devices that generated more than 300 dialogs, were tagged with more than one tree representation. We performed a multi-class classification using the one versus all method described in [41]. The classification precision was 80 % which is a relative promising result. This result was obtained using a 5 fold validation technique - one fifth of the data was taken out and used to assess the accuracy/precision of the system. The remaining four fifths of data was used to train the system. Table 1 summarizes a subset of the SIP devices used for training/testing. We could not include the complete table in the paper due to space constraints. However, the different columns of table 1 give a glance of the data samples and associated tree structures.

Table 1. Tested equipment

Device	#Msgs	#Sessions	#Dialogs	#Nodes	Depth
TrixboxCE_v2.6.0.7	1935	714	544	279	99
Twinkle_v1.1	421	129	109	146	36
Thomson2030_v1.59	345	102	83	105	34
Cisco-7940_v8.9	457	175	139	54	18
Linksys_v5.1.8	397	130	67	206	99
SJPhone_v1.65	627	246	210	66	19

For instance, the Tribox CE (a popular VoIP PBX) has a tree representation of depth 99 and 279 nodes. This was learned using 1935 messages split over 544 SIP dialogs.

7 Conclusion and Future Work

We have addressed in this paper the automated fingerprinting of unknown protocols. Our approach is based on the unsupervised learning of the types of messages that are used by actual implementations of that protocol. The unsupervised learning method relies on support vector clustering - SVC. Our technique is using a new metric - the weighted character position metric. This metric is computed rapidly and does not suppose any knowledge about the protocols: header fields specification, number of messages. One main advantage of the SVC technique is its improvement of the accuracy of the classification for large datasets. We have also proposed a semi automated method that allows to choose the best parameters. The observed message types can be used to induce a tree-like representation of the underlying state machines. The nodes in this tree represent the different types of observed messages and the edges do indicate an invocation relationship between the nodes. This first phase is completed by a second stage, where the behavioral differences are extracted and mined. This second phase uses tree kernel support vector machines to model the finite state machines induced from the first phase. The main novelty of this approach lies in the direct usage and mining of the induced state machines. We did test our approach on extensive datasets for several well known protocols: SIP, SMTP and IMAP. The observed empirical accuracy is very good and promising. We plan to extend this work towards other machine learning tasks and conceptual solutions. In addition, finding specific metrics for encrypted and binary protocols is another direction for future work.

Acknowledgments. We would like to thank Yann Guermeur, researcher at CNRS, for his support and feedback on the SVC specific implementation. This work was partially supported by the French National Research Agency under the VAMPIRE project ref#ANR-08-VERS-017.

References

1. Tridgell, A.: How samba was written, http://samba.org/ftp/tridge/misc/french_cafe.txt (accessed on 03/16/09)
2. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through conectect-aware monitored execution. In: 15th Symposium on Network and Distributed System Security, NDSS (2008)
3. Cui, W., Paxson, V., Weaver, N., Katz, R.H.: Protocol-independent adaptive replay of application dialog. In: Symposium on Network and Distributed System Security, NDSS (2006)
4. Leita, C., Mermoud, K., Dacier, M.: Scriptgen: an automated script generation tool for honeyd. In: Computer Security Applications Conference, Annual, pp. 203–214 (2005)
5. Arkin, O.: Icmp usage in scanning: The complete know-how, version 3 (June 2001) (accessed on 03/16/09)
6. tcpdump, http://www.tcpdump.org/ (accessed on 02/05/09)
7. Beddoe, M.: Protocol informatics, http://www.4tphi.net (accessed on 02/05/09)
8. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: CCS 2008: Proceedings of the 15th ACM conference on Computer and communications security, pp. 391–402. ACM, New York (2008)
9. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: CCS 2007: Proceedings of the 14th ACM conference on Computer and communications security, pp. 317–329. ACM, New York (2007)
10. Gopalratnam, K., Basu, S., Dunagan, J., Wang, H.J.: Automatically extracting fields from unknown network protocols (June 2006)
11. Weidong: Discoverer: Automatic protocol reverse engineering from network traces, pp. 199–212
12. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium, NDSS 2008 (2008)
13. Shevertalov, M., Mancoridis, S.: A reverse engineering tool for extracting protocols of networked applications, October 2007, pp. 229–238 (2007)
14. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: automatic protocol replay by binary analysis. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 311–321. ACM, New York (2006)
15. Comer, D., Lin, J.C.: Probing TCP Implementations. In: USENIX Summer, pp. 245–255 (1994)
16. P0f, http://lcamtuf.coredump.cx/p0f.shtml
17. Nmap, http://www.insecure.org/nmap/
18. Caballero, J., Venkataraman, S., Poosankam, P., Kang, M.G., Song, D., Blum, A.: FiG: Automatic Fingerprint Generation. In: The 14th Annual Network & Distributed System Security Conference (NDSS 2007) (February 2007)
19. Scholz, H.: SIP Stack Fingerprinting and Stack Difference Attacks. Black Hat Briefings (2006)
20. Yan, H., Sripanidkulchai, K., Zhang, H., yin Shae, Z., Saha, D.: Incorporating Active Fingerprinting into SPIT Prevention Systems. In: Third Annual VoIP Security Workshop (June 2006)

21. Ma, J., Levchenko, K., Kreibich, C., Savage, S., Voelker, G.M.: Unexpected means of protocol inference. In: Almeida, J.M., Almeida, V.A.F., Barford, P. (eds.) Internet Measurement Conference, pp. 313–326. ACM, New York (2006)
22. Haffner, P., Sen, S., Spatscheck, O., Wang, D.: Acas: automated construction of application signatures. In: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data (Minet), pp. 197–202. ACM, New York (2005)
23. Abdelnur, H.J., State, R., Festor, O.: Advanced Network Fingerprinting. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 372–389. Springer, Heidelberg (2008)
24. Crocker, D., Overell, P.: Augmented BNF for Syntax Specifications: ABNF. RFC 2234 (Proposed Standard) (1997)
25. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), Updated by RFCs 3265, 3853, 4320, 4916, 5393 (2002)
26. Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), Updated by RFC 5506 (2003)
27. Krügel, C., Toth, T., Kirda, E.: Service specific anomaly detection for network intrusion detection. In: SAC 2002: Proceedings of the 2002 ACM symposium on Applied computing, pp. 201–208. ACM, New York (2002)
28. Ben-hur, A., Horn, D., Siegelmann, H.T., Vapnik, V.: Support vector clustering. Journal of Machine Learning Research 2, 125–137 (2001)
29. Day, W.H., Edelsbrunner, H.: Efficient algorithms for agglomerative hierarchical clustering methods. Journal of Classification 1(1), 7–24 (1984)
30. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)
31. Wang, L. (ed.): Support Vector Machines: Theory and Applications. Studies in Fuzziness and Soft Computing, vol. 177. Springer, Heidelberg (2005)
32. Berkhin, P.: A survey of clustering data mining techniques. In: Grouping Multidimensional Data, pp. 25–71 (2006)
33. Klensin, J.: Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), Obsoleted by RFC 5321, updated by RFC 5336 (April 2001)
34. Crispin, M.: Internet Message Access Protocol - Version 4rev1. RFC 3501 (Proposed Standard), Updated by RFCs 4466, 4469, 4551, 5032, 5182 (March 2003)
35. Salvador, S., Chan, P.: Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In: ICTAI 2004: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, USA, pp. 576–584. IEEE Computer Society, Los Alamitos (2004)
36. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. In: STOC 1989: Proceedings of the twenty-first annual ACM symposium on Theory of computing, pp. 411–420. ACM, New York (1989)
37. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
38. Schapire, R.E.: Diversity-based inference of finite automata. Technical report, Cambridge, MA, USA (1988)
39. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron. In: ACL 2002: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, Morristown, NJ, USA, pp. 263–270 (2002)
40. Vishwanathan, S., Smola, A.: Fast kernels on strings and trees. In: Proceedings of Neural Information Processing Systems (2002)

41. Moschitti, A.: Making tree kernels practical for natural language learning. In: Proceedings of the Eleventh International Conference on European Association for Computational Linguistics (2006)
42. Moschitti, A.: Efficient convolution kernels for dependency and constituent syntactic trees. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 318–329. Springer, Heidelberg (2006)
43. Moschitti, A., Pighin, D., Basili, R.: Tree kernel engineering for proposition re-ranking. In: Proceedings of Mining and Learning with Graphs, MLG 2006 (2006)
44. Moschitti, A.: M-light-tk 1.2 (feature vector set and tree forest) (2009)

SMS-Watchdog: Profiling Social Behaviors of SMS Users for Anomaly Detection

Guanhua Yan, Stephan Eidenbenz, and Emanuele Galli

Information Sciences (CCS-3)*
Los Alamos National Laboratory
{ghyan,eidenben,egalli}@lanl.gov

Abstract. With more than one trillion mobile messages delivered worldwide every year, SMS has been a lucrative playground for various attacks and frauds such as spamming, phishing and spoofing. These SMS-based attacks pose serious security threats to both mobile users and cellular network operators, such as information stealing, overcharging, battery exhaustion, and network congestion. Against the backdrop that approaches to protecting SMS security are lagging behind, we propose a lightweight scheme called *SMS-Watchdog* that can detect anomalous SMS behaviors with high accuracy. Our key contributions are summarized as follows: **(1)** After analyzing an SMS trace collected within a five-month period, we conclude that for the majority of SMS users, there are window-based regularities regarding whom she sends messages to and how frequently she sends messages to each recipient. **(2)** With these regularities, we accordingly propose four detection schemes that build normal social behavior profiles for each SMS user and then use them to detect SMS anomalies in an online and streaming fashion. Each of these schemes stores only a few states (typically, at most 12 states) in memory for each SMS user, thereby imposing very low overhead for online anomaly detection. **(3)** We evaluate these four schemes and also two hybrid approaches with realistic SMS traces. The results show that the hybrid approaches can detect more than 92% of SMS-based attacks with false alarm rate 8.5%, or about two thirds of the attacks without any false alarm, depending on their parameter settings.

Keywords: SMS, anomaly detection, relative entropy, JS-divergence.

1 Introduction

The Short Message Service (SMS) provided by cellular carriers is a connectionless message transfer service with low capacity. Since its inception in December 1992, when the first short message was delivered in the Vodafone GSM network in the United Kingdom [12], SMS has been growing at a blistering speed. According to the IDC research firm, the total number of SMS subscribers in the US in 2006 was estimated at 102 million and is expected to reach 184 million in 2011; the number of short messages delivered in the US will grow at an even faster pace,

* Los Alamos National Laboratory Publication No. LA-UR 08-07637.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 202–223, 2009.
© Springer-Verlag Berlin Heidelberg 2009

which will jump from 157 billion in 2006 to 512 billion in 2011 [22]. Another report by Gartner shows that the Asia-Pacific region, whose SMS subscribers sent 1.5 trillion short messages in 2007, is leading the SMS growth over North America and Western Europe, whose SMS subscribers generated 189 and 202 billion short messages in 2007, respectively [18].

Due to its increasing popularity, SMS has become a lucrative target for fraudulent behaviors that have been rampant in the Internet for decades. For instance, 88 percent of mobile users in China [4] have been plagued by SMS spams. Meanwhile, the convergence of the telecommunication world and the Internet has led to the emergence of SMS phishing, also dubbed "SMiShing", which could steal confidential account information from mobile devices [16] or spreading mobile malware [15]. SMS spoofing is another type of attacks based on SMS: by manipulating address information in SMS messages, an SMS spoofing attack simulates the behavior of a legitimate mobile device so that foreign networks (as opposed to the home network of the mobile device) mistakingly think these messages originate from that device. SMS spoofing attacks have been launched against major cellular operators in Europe in the past [19]. Besides these spamming, phishing and spoofing attacks, other SMS-based attacks include SMS flooding, which aims to overload the cellular network, and SMS faking, which mimics the behavior of an SMS switch to send messages [17]. In particular, SMS flooding could shut down cellular services entirely in a large area [6].

SMS-based attacks pose serious security threats to both mobile users and cellular networks, including information stealing, overcharging, battery exhaustion, and network congestion. Effective countermeasures, unfortunately, are still lagging behind. Many existing solutions are aimed at detecting malware on mobile devices with techniques inspired by their counterparts in IP networks. For instance, signature-based detection schemes are proposed to examine mobile network traffic [8] or power usage of mobile applications [9] for signatures that are extracted from existing mobile malware instances. A machine learning-based approach is developed in [1] to catch mobile malware by discriminating behaviors of normal applications and malware at the level of system events and API calls.

Although effective against some SMS-based attacks, these mobile malware detection approaches have their flip sides. *First*, not all aforementioned SMS-based attacks originate from mobile malware. For instance, SMS spoofing usually comes from a device that simulates an authentic mobile handheld to fool a foreign network. *Second*, many of these approaches demand extra computational resources on mobile devices, which accelerate exhaustion of their batteries. *Third*, operating systems of existing mobile devices often lack sophisticated countermeasures to prevent mobile malware from disabling device-resident detection schemes.

In this work, we take a different avenue to detect anomalous SMS behaviors. We propose a detection framework called *SMS-Watchdog*, which is deployed at a place where a mobile user's short message records can be easily accessed, such as the Short Messaging Service Center (SMSC) in a typical SMS architecture. Hence, our work alleviates typical shortcomings of device-resident mobile detection schemes, such as extra power consumption for detection and inability to

catch spoofed short messages. Motivated by observations made from a real-world SMS dataset, our work exploits regularities inherent in a typical user's SMS behaviors for anomaly detection. Our key contributions in this paper are summarized as follows: **(1)** After analyzing an SMS trace collected within a five-month period, we conclude that for the majority of SMS users, there are window-based regularities regarding whom she sends messages to and how frequently she sends messages to each recipient. **(2)** With these regularities, we accordingly propose four detection schemes that build normal social behavior profiles for each SMS user and then use them to detect SMS anomalies in an online and streaming fashion. Each of these schemes stores only a few states (typically, at most 12 states) in memory for each SMS user, thereby imposing very low overhead for online anomaly detection. **(3)** We evaluate these four schemes and also two hybrid approaches with realistic SMS traces and the results show that the hybrid approaches can detect more than 92% of SMS-based attacks with false alarm rate 8.5%, or two thirds of the attacks without any false alarm, depending on their parameter settings.

Related work. In [27], Zerfos et al. used an SMS trace collected from a national cellular carrier in India to examine message size distribution, message service time distribution, and thread-level characteristics. The same trace was later investigated by Meng et al. to understand delivery reliability and latency [13]. It is noted that their trace, although containing more SMS users than the one analyzed in this paper, lasts only three weeks. By contrast, our trace contains short messages within five months, thus offering insights on long-term and persistent behaviors of SMS users. Enck et al. showed that SMS flooding from the Internet side can cause severe denial-of-service attacks against cellular network operations in a large area [6] and later they proposed some countermeasures to prevent such attacks [24]. Our work in this paper attempts to identify anomalous SMS behaviors by comparing them against normal SMS user's social behavior profiles and is thus complementary to their work.

Techniques based on profiling human social behaviors have been applied to detect anomalies in other types of network traffic. For instance, Stolfo et al. developed a data mining system that builds behavioral profiles or models with methods such as user cliques, Hellinger distance, and cumulative distributions for emails users [20]. Using injected simulated viral emails, they show that the detection system performs well with high accuracy. In [26], Yan et al. applied change-point detection techniques to detect worm propagation in IM (Instant Messaging) networks. They show that schemes simply counting the number of instant messages sent by each user can easily be circumvented by a carefully crafted IM worm. Observing that the distribution of the number of messages sent to each contact on the buddy list is typically highly skewed, they developed an effective technique that periodically calculates the overall likelihood that instant messages are sent to each recipient and look for abrupt changes to detect IM worm propagation. Moreover, in this work we use information-theoretical measures to detect anomalies in SMS traffic and these techniques have been applied to detect anomalies in Internet traffic before [11][14]. Besides focusing on a

different type of network traffic, we also address the scalability issues that were not considered in previous work. For a more comprehensive survey on anomaly detection techniques and their applications, we refer interested readers to [2].

Anomaly detection for mobile phone networks has a long history. For instance, calling activities have been examined to detect mobile phone fraud [7][23][5] and mobility patterns of mobile devices have been profiled to detect cloning attacks and cell phone losses [25][21]. Recently, due to the increasing popularity of smart phones, a growing number of malware instances have been observed on these mobile devices. Many existing approaches to detect mobile malware work by profiling behaviors of normal applications [9][1]; our work instead focuses on profiling normal SMS user behaviors.

Organization. The remainder of the paper is organized as follows. Section 2 briefly introduces typical SMS architectures and how we collect the SMS trace. In Section 3, we first analyze SMS user behaviors in the trace and discuss what statistic metrics exhibit low variation. We then discuss the design of SMS-Watchdog in Section 4. Based on the results from the trace analysis in Section 3, we propose four different detection schemes in Section 5. We evaluate the performance of these four schemes and also two hybrid schemes in Section 6. We finally make concluding remarks and discuss the scope of our work in Section 7.

2 Background on SMS and SMS Traces

SMS architecture. SMS is a service that provides a connectionless transfer of messages with at most 160 characters using signaling channels in cellular networks. Figure 1 illustrates the basic SMS architecture in a GSM-based system. A short message sender (on the right bottom corner) uses an *originating MS* (Mobile Station) to send a short message to a receiver (on the left bottom corner in Figure 1). The short message is delivered to a nearby BSS (Basestation System) through the GSM signaling channel and then the MSC (Mobile Switching Center) associated with this BSS. The MSC first checks with a VLR (Visitor Location Register) database whether the originating MS is allowed to receive

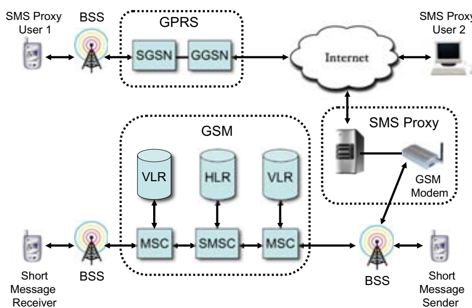


Fig. 1. SMS Architecture and SMS Proxy

the short message service. The VLR database temporarily stores subscription information for the visiting mobile stations so that the associated MSC knows what services should be provided to them. If the originating MS is allowed to use SMS, the MSC further routes the short message to SMSC, a dedicated store-and-forward server that handles SMS traffic.

The SMSC is responsible for forwarding the short message to the targeted mobile device, also called *terminating MS* (on the left bottom corner). To do that, it queries an HLR (Home Location Register) database, which keeps information about cellular subscribers, such as their profile information, current location, billing data, and validation period. The HLR responds by sending back the serving MSC address of the terminating MS. Thereafter, the SMSC forwards the short message to that MSC, which further queries its associated VLR database for the location area of the terminating MS. Once the location of the terminating MS is found, the short message is delivered to it through its nearby BSS.

SMS trace collection. The SMS trace used in this work was collected from an SMS proxy in Italy. This SMS proxy connects with the Internet, and also the GSM network through a GSM modem. There are two classes of users for this SMS proxy. Similar to the regular SMS sender, the first class (e.g., User 1) also use an MS to send short messages. These short messages, however, do not use GSM signaling channels; instead, they are delivered through the GPRS network to the SMS proxy, which further forwards these messages to their recipients through the GSM modem. The second class of SMS proxy customers (e.g., user 2) send their short messages to the SMS proxy through the Internet and then to their receiving MSes through the GSM modem. The economic incentive for such an SMS proxy is the price difference between regular SMS and GPRS messages.

The SMS proxy was launched in early 2008 and we obtained communication logs of all its users from April 15, 2008 to September 14, 2008. Through this period, there were 2,121 users that have used it to send short messages. In total, 146,334 short messages have been sent through this proxy. As this data trace covers a large number of users, we believe that it is representative of general SMS traffic. Due to its short history, the SMS proxy has not been seen suffering malicious attacks yet. Hence, analysis on the data collected sheds light on how *normal* SMS users behave socially. Moreover, the long time span of this trace enables us to investigate the persistent behavioral patterns of SMS users. This is contrast to previous work which mainly focus on analyzing communication traces of SMS users only within a short period of time [27][13].

3 Trace Analysis

In this section, we analyze the dynamics of the system from which the trace was collected and then derive regularities inherent in behaviors of normal SMS users.

System dynamics. In Figure 2, we show the number of short messages that have been observed by the SMS proxy each day. From the graph, we observe an

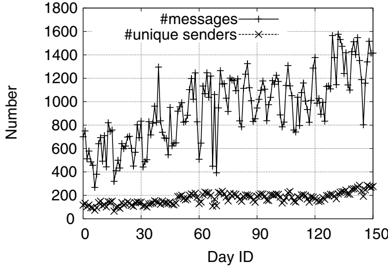


Fig. 2. Number of messages and number of unique senders per day

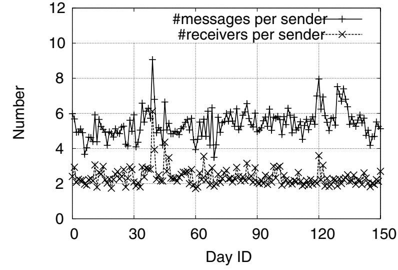
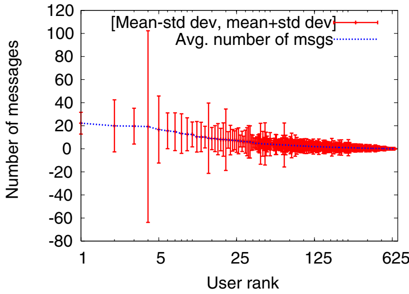
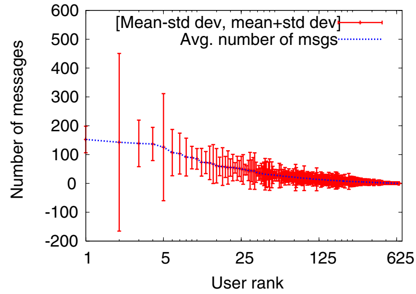


Fig. 3. Number of messages and receivers per sender per day



(1) Daily



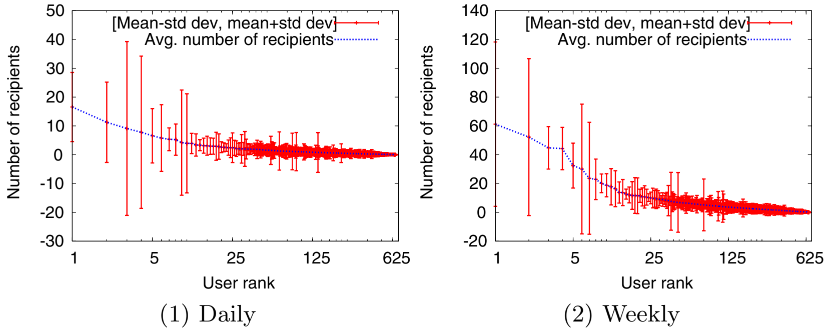
(2) Weekly

Fig. 4. Average numbers of messages for persistent users

obvious trend that an increasing number of short messages have been transmitted through the SMS proxy. For instance, the number of short messages observed has increased by 67% from May to August. This is attributed to the growing number of users of the system during the trace collection period. Figure 2 also depicts the number of users that sent at least one message in a day. In August, there were 972 active customers, as opposed to only 662 ones in May.

Although the number of users in the system was not stationary during the data collection period, we find that both the number of messages sent out by each user and the number of receivers per sender each day are quite stationary. They are illustrated in Figure 3, from which we observe that each day an active user sends about 5.4 messages to 2.4 recipients on average. We also note that some users in the dataset used the SMS proxy to send short messages for only a short period of time. As we are only interested in *persistent* behaviors of SMS users, we do not consider these temporary users. From the dataset, we obtain a list of 662 users whose first and last short messages were sent at least 60 days apart. For brevity, we call them *persistent users*, who contributed about 75% of the entire set of short messages.

Temporally periodic behaviors of persistent users. We are interested in statistically time-invariant metrics that characterize behaviors of SMS users so

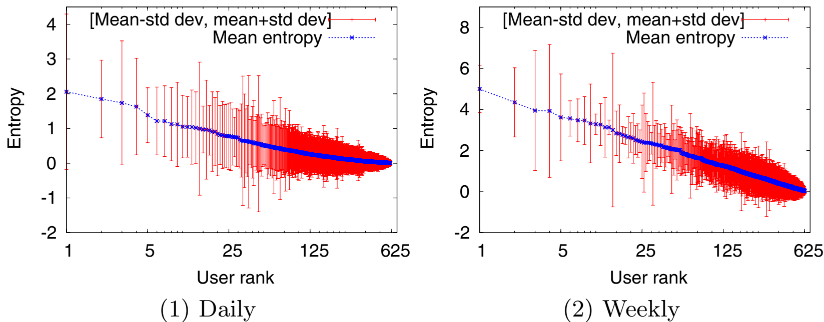


(1) Daily (2) Weekly

Fig. 5. Average numbers of recipients for persistent users

that they can be applied for anomaly detection. An ideal metric should have low variation, which helps reduce false alarm rates. As human behaviors such as lunch and shopping often exhibit temporal periodicity, a natural hypothesis is that SMS user behaviors should follow similar patterns. To verify this, we depict in Figure 4 the average number of messages sent out per day and per week by each persistent user versus his overall rank. Note that the x-axis, which indicates the user rank, is shown in logarithmic scale. For each persistent user in the graphs, we also show the range between the mean plus and minus one standard deviation. Clearly, these graphs reveal that both daily and weekly numbers of messages sent by persistent users exhibit high variation for many users.

A better way of quantifying the variation of a statistic metric is *Coefficient of Variation* (COV), defined as the ratio of the standard deviation to the mean. Generally speaking, a distribution with $COV < 1$ portends low variation, and those with $COV > 1$ are considered high variation. Among all the persistent users, 97.7% and 71.9% of them have COVs > 1 for the daily and weekly number of short messages they sent, respectively, suggesting that neither of these two metrics is good for anomaly detection.



(1) Daily (2) Weekly

Fig. 6. Average entropies for persistent users

We perform a similar analysis on the daily and weekly number of unique recipients to whom persistent users sent their short messages, and the results are provided in Figure 5. We observe that 94.4% and 54.4% of persistent users have COVs > 1 for their daily and weekly number of unique recipients, respectively. Hence, they are not good candidates for anomaly detection either.

We further analyze the *entropy* of the distribution of the number of short messages sent to each unique recipient for every persistent SMS user. The entropy, H, is defined as: $H = -\sum_{i=1}^{n} p_i \times \log_2 p_i$, where p_i is the fraction of short messages sent to the i-th unique recipient in a day or a week (suppose that there are n unique recipients). Figure 6 shows the average daily and weekly entropies for each persistent SMS user. Similar to the other metrics that we have studied, these two also show high variation: 98.0% and 66.9% of persistent users have COVs > 1 for their daily and weekly entropies, respectively. Therefore, neither metric seems plausible for anomaly detection.

Window-based behaviors of SMS users. The above analysis reveals that high variation is inherent in many SMS users' behaviors on a temporally periodic basis. We now examine their behaviors from a window-based perspective. For each SMS user in the dataset, we form m blocks, each of which contains an equal number of successive short messages. Given the sequence of blocks from the same SMS sender, we first consider the number of unique recipients to whom messages in each block are sent. Similar to our previous analysis, we are interested in the variation of this metric. To ensure that there are enough short messages in each block, we consider only users that have sent at least θ short messages. In our study, we consider two θ values, 100 and 200, which lead to a set of 353 and 167 qualified SMS users, respectively.

Figure 7 gives the average number of unique recipients when θ is 200 (we have similar results for $\theta = 100$, but due to space limitation, we do not show them here). Different from the metrics characterizing temporally periodic behaviors, the number of unique recipients seen in each window seems to have low variation. This is confirmed by Table 1, which shows that the COV exceeds 1.0 for less than 1% of the users, regardless of which θ and m are used.

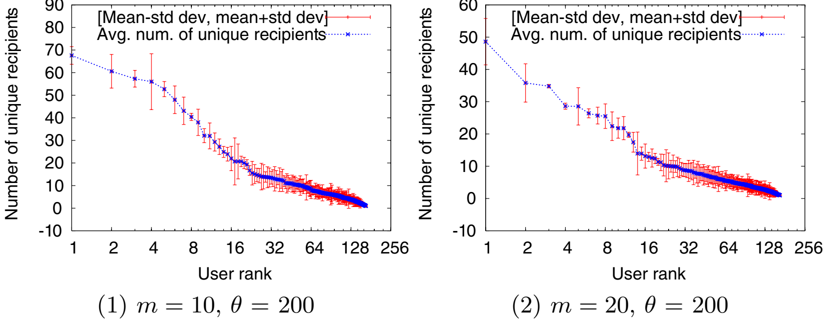
For each SMS user, we also consider the entropy of the distribution of the number of messages sent to each unique recipient within each block. Figure 8 depicts the mean for $\theta = 200$, and Table 1 also provides the average COV in different combinations of θ and m. In all cases, the average COV is smaller than 20%. It also seems that the COV can be reduced by either increasing the threshold θ or choosing a smaller m.

These results reveal that window-based behaviors of SMS users bear lower variation than their temporally periodic behaviors. In the following discussion, we further explore the similarity across different blocks for each SMS user.

Similarity measures. First, we study the similarity in the set of recipients between different blocks for the same SMS user. For the i-th block $\mathcal{B}_i$ associated with an SMS user, we let $\mathcal{R}_i$ denote the entire set of unique recipients of the short messages in this block. We use the following *recipient similarity metric* to

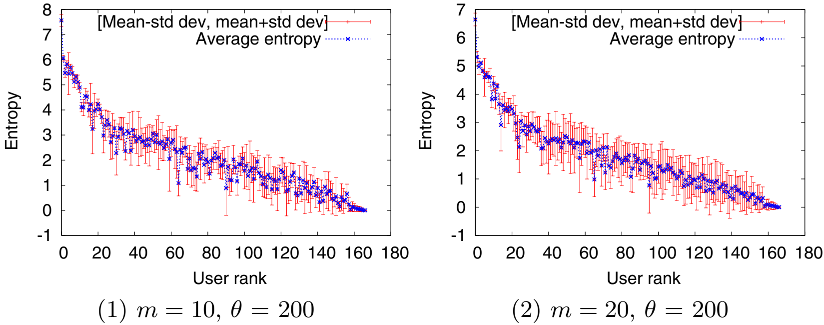
Table 1. Fraction of users with COV > 1 regarding window-based behaviors

m	$\theta = 100$		$\theta = 200$	
	#Recipients	Entropy	#Recipients	Entropy
10	0.3%	14.5%	0.6%	11.2%
20	0.3%	18.1%	0.6%	15.6%



(1) $m = 10$, $\theta = 200$ (2) $m = 20$, $\theta = 200$

Fig. 7. Average number of unique recipients



(1) $m = 10$, $\theta = 200$ (2) $m = 20$, $\theta = 200$

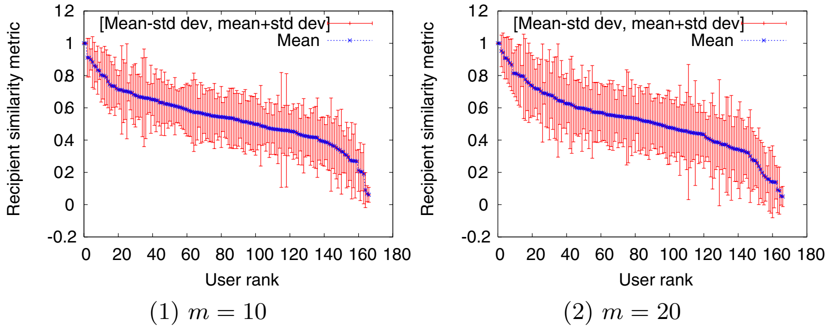
Fig. 8. Average entropies

measure the distance between two sets $\mathcal{R}_i$ and $\mathcal{R}_j$ ($i \neq j$):

$$S(\mathcal{R}_i, \mathcal{R}_j) = \frac{|\mathcal{R}_i \cap \mathcal{R}_j|}{2} \times (\frac{1}{|\mathcal{R}_i|} + \frac{1}{|\mathcal{R}_j|}). \tag{1}$$

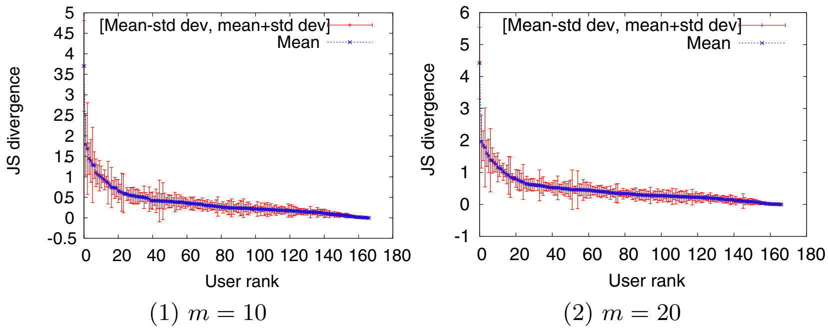
$S(\mathcal{R}_i, \mathcal{R}_j)$ gives the average fraction of common elements that are shared between sets $\mathcal{R}_i$ and $\mathcal{R}_j$. Clearly, $S(\mathcal{R}_i, \mathcal{R}_j)$ close to 0 means that $\mathcal{R}_i$ and $\mathcal{R}_j$ share few common elements and vice versa if it is close to 1. For each SMS user, we call set $\{S(\mathcal{R}_i, \mathcal{R}_j) : \forall i, j, i \neq j\}$ as her *recipient similarity metric set*.

Figure 9 depicts the recipient similarity metrics with $\theta = 200$. One observation is that SMS users differ significantly on how they send messages regularly: for some users they send short messages to almost the same set of recipients, but for



(1) $m = 10$ (2) $m = 20$

Fig. 9. Recipient similarity metric for all recipients ($\theta = 200$)



(1) $m = 10$ (2) $m = 20$

Fig. 10. JS-divergence for all recipients ($\theta = 200$)

Table 2. Fraction of users with COV > 1 regarding similarity measures for all recipients

m	$\theta = 100$		$\theta = 200$	
	Recipient similarity	JS divergence	Recipient similarity	JS divergence
10	4.5%	4.3%	0.6%	1.2%
20	12.8%	5.1%	6.0%	1.2%

some others[1] they send short messages to a very diverse set of recipients. Given this fact, we thus *cannot* conclude that SMS users always tend to send short messages to the same set of users over time, which, if true, would be useful for anomaly detection. We further analyze the variation of the recipient similarity metric set for each SMS user and the results are given in Table 2. Interestingly, this measure exhibits low variation for the majority of the SMS users, implying that the recipients to whom most SMS users send short messages vary in a very similar fashion over time.

[1] These users typically send bulk messages for advertisement purposes.

Previously we have used the entropy to measure the uncertainty of the distribution of the number of short messages sent to each recipient. A natural question extended from that is how similar these distributions are across different messages blocks. A metric commonly used for this is *relative entropy*, also called *Kullback-Leibler (KL) divergence* [3], which is defined as follows:

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}, \tag{2}$$

where P and Q are two distributional functions.

Note that $D_{KL}(P\|Q)$ is undefined if the support[2] of P is not a subset of the support of Q. Hence, directly applying the relative entropy here is problematic because an SMS user may have different sets of recipients in two message blocks. Moreover, relative entropy is not symmetric, i.e., $D_{KL}(P\|Q)$ may not equal $D_{KL}(Q\|P)$. Due to these constraints, we instead use the *Jensen-Shannon (JS) divergence* [10], whose computation relies on the KL-divergence:

$$D_{\mathrm{JS}}(P\|Q) = \frac{1}{2}[D_{\mathrm{KL}}(P\|\tilde{A}_{P,Q}) + D_{\mathrm{KL}}(Q\|\tilde{A}_{P,Q})], \tag{3}$$

where function $\tilde{A}_{P,Q}$ denotes the average distribution: $\tilde{A}_{P,Q}(x) = (P(x)+Q(x))/2$. Obviously, $D_{\mathrm{JS}}(P\|Q)$ is always defined and also symmetric.

Figure 10 shows the JS-divergences when $\theta = 200$. We observe that the JS-divergence also varies significantly among different SMS users. It is clear that the JS-divergence has low variation for the majority of the SMS users, which is confirmed by Table 2: for all four combinations of θ and m, less than 6% percent of the SMS users have a JS-divergence COV greater than 1.

Next, we show how the top five recipients receiving the most messages correlate with each other between different blocks. Figure 11 depicts the recipient similarity metrics for the top 5 recipients with $\theta = 200$. Still, this metric varies significantly among different SMS users. The fractions of SMS users with $COV > 1$ are shown in Table 3. Clearly, these fractions are higher than their counterparts where all recipients are considered, but are still relatively small.

Similarly, we study the JS-divergence of the distributions of the numbers of messages sent to the top five recipients among different blocks. Here, we normalize the probability that *each* of those top five recipients receives a message by dividing it by the probability that *any* of those top five recipients receives

Table 3. Fraction of users with COV > 1 for top-5 recipients (similarity measure)

m	$\theta = 100$		$\theta = 200$	
	Recipient similarity	JS divergence	Recipient similarity	JS divergence
10	8.2%	7.9%	6.6%	5.4%
20	14.4%	13.6%	9.6%	12.6%

[2] The *support* of a function is the set of points where the function is not zero.

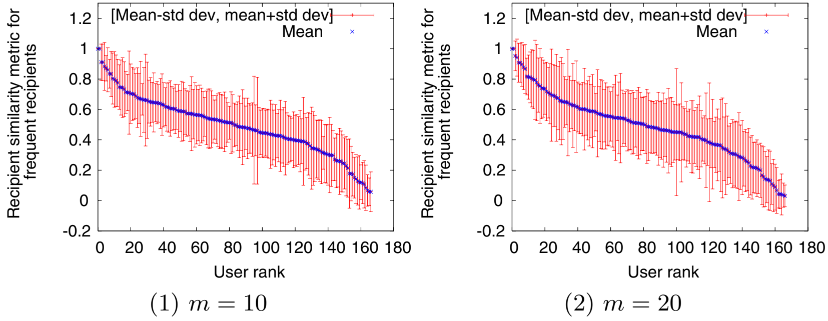


Fig. 11. Recipient similarity metric for top 5 recipients ($\theta = 200$)

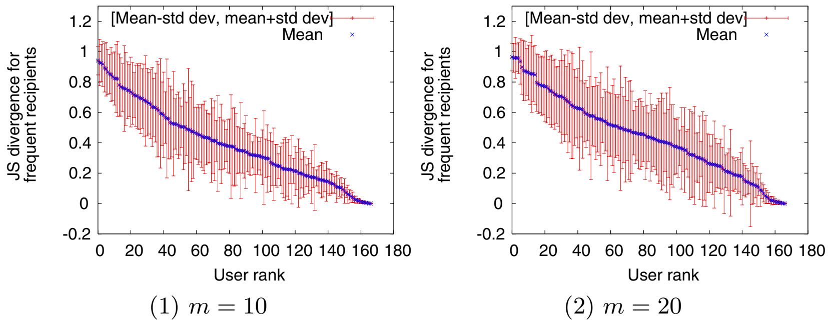


Fig. 12. JS-divergence for top 5 recipients ($\theta = 200$)

a short messages. For instance, if the top five recipients receive 5, 4, 3, 2, and 1 short messages within a window, the normalized probabilities are 1/3, 4/15, 1/5, 2/15, and 1/15, respectively.

The JS-divergence of the distributions for the top five recipients is shown in Figure 12 for $\theta = 200$. We notice that the average JS-divergence is always no higher than 1 and the fractions of SMS users with COV > 1 exhibit a similar pattern as the recipient similarity metric: although they are higher than their counterparts where all recipients are considered, they are still very small.

In summary, different SMS users may have different levels of similarity across their message blocks, but the level of similarity across different message blocks of *the same* SMS user typically does not change significantly.

4 SMS-Watchdog Design

In the following, we shall discuss how to exploit the regularities inherent in social behaviors of SMS users for anomaly detection. Before presenting the detailed algorithms, we first discuss two families of SMS-related attacks that are considered in this work and then present the design of SMS-Watchdog.

Threat model. Two families of SMS-related attacks are considered here. The first type is called *blending attacks*, which occur when an SMS user's account is used to send messages for a different person. In reality, this can happen in three circumstances. First, a user's cell phone is implanted with a Trojan horse such that the cell phone can be remotely controlled to send messages for a different user[3]. Second, in an SMS spoofing attack, a fraudster can manipulate address information in messages to spoof a legitimate SMS user's identity in a foreign network. Third, if an SMS proxy is used (e.g., the one in Figure 1), an attacker can hack an SMS user's account at the front end and use it to send messages. All these attacks are termed as blending attacks because illegitimate messages are intermingled with legitimate ones from the detector's perspective. The second type of attacks, termed as *broadcast attacks*, mirrors the behavior of mobile malware that send out phishing or spamming messages to recipients that appear in normal ones. In such attacks, the mobile device from which these messages are sent have already been infected by the mobile malware.

Workflow of SMS-Watchdog. In our design, the SMS-Watchdog is placed at the SMSC, as shown in the SMS architecture in Figure 1, which handles all SMS traffic for a specific cellular network. The workflow of SMS-Watchdog involves three steps:

(1) Monitoring: SMS-Watchdog maintains a detection window of size h for each SMS user that has subscribed for this service. For the current detection window, it also keeps a counter k for the number of sent SMS messages observed, and the sequence of recipients of these SMS messages. When k becomes equal to h, SMS-Watchdog performs anomaly detection for this user as shown in Step *(2)*.

(2) Anomaly detection: Given the recipients of the last h SMS messages, the SMS-Watchdog checks whether there exist anomalous behaviors. If so, it raises an alert and goes to the next step. The detailed algorithms for anomaly detection will be presented in the next section.

(3) Alert handling: SMS-Watchdog sends an alert to the SMS user through a different communication channel, such as emails. Together with the alert, SMS-Watchdog also sends to the user a summary of the last h messages, such as the number of SMS messages per recipient and the whole time frame of these messages. The information is used to help the user to identify false positives. The user can first check whether the communication record shown on her mobile device matches with the summary sent by SMS-Watchdog within the given time frame[4]. If the two do not match, it means that the user's SMS account has been spoofed and she can notify her service provider. Otherwise, the user further checks the summary to identify suspicious SMS behaviors. A more cautious user can even request to check the full communication record, regarding the

[3] To evade detection, the malware can delete the message from the "sent" folder after it is sent out; also, the message can attach a different returning number so that the recipient will not reply the message to the compromised phone.

[4] This can be automatically done with a software for the user's convenience.

transmission time of each SMS message. If suspicious SMS behaviors have been observed, it is likely that the mobile device has been infected by malware and the user can use some anti-virus software to disinfect her mobile device.

5 Anomaly Detection

In this section, we provide the details on how Blue-Watchdog performs anomaly detection. The anomaly detection problem is formulated as follows: *given an SMS user's communication history* $\mathcal{V} = \{v_1, v_2, ..., v_n\}$*, where* $v_i (1 \leq i \leq n)$ *denotes the recipient of the* i*-th short message in* $\mathcal{V}$*, and a test sequence* $\mathcal{T} = \{t_1, t_2, ..., t_h\}$*, where* t_j $(1 \leq j \leq h)$ *denotes the recipient of the* j*-th short message in* $\mathcal{T}$*, is* $\mathcal{T}$ *anomalous?*

Decision on detection window size h. We first address how to choose h, the detection window size for a specific user. As revealed in Section 3, a typical SMS user's window-based behaviors bear low variation in the number of unique recipients, entropy, recipient set similarity metric, and also JS-divergence, suggesting that choosing h based on any of these metrics would be a possible solution. Compared with the other three metrics, however, the JS-divergence contains the most information, because its calculation depends on not only the set of recipients, the distribution of the number of short messages sent to each recipient, but also the distances between these distributions.

A feasible choice for h is minimizing the COV of the JS-divergence after grouping sequence $\mathcal{V}$ by every h short messages, because this can maximize the level of similarity among different blocks. Let $cov(X)$ denote the COV of set X. We choose h^* as follows:

$$h^* = \mathop{\mathbf{argmin}}_{h_{min} \leq h \leq h_{max}} cov(\{D_{JS}(P_i \| P_j) \mid 1 \leq i < j \leq \lfloor \frac{n}{h} \rfloor\})$$

where P_s $(1 \leq s \leq \lfloor \frac{n}{h} \rfloor)$ is the distribution of the number of short messages sent to each unique recipient within block $[v_{(s-1)h+1}, v_{(s-1)h+2}, ..., v_{sh}]$.

It is important to bound h from both sides. On one hand, we need to ensure that the training sequence $\mathcal{V}$ is split into enough blocks so that the COV does not approach 0 (note that if $h = |\mathcal{V}|$, the COV is always 0). Also, a large h induces high detection delay. On the other hand, we choose a sufficiently large h to avoid performing anomaly detection too frequently. Hence, we constrain the selection of h between h_{min} and h_{max}, both of which are configurable parameters.

Mean-based anomaly detection. As both the average number of unique recipients and the average entropy within each block show low variation for most SMS users, the mean-based anomaly detection scheme identifies anomalous SMS behavior by checking whether the means of these two metrics in the test sequence $\mathcal{T}$ deviate significantly from the means observed from the history trace $\mathcal{V}$. Let $\{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_l\}$, where $l = \lfloor \frac{n}{h^*} \rfloor$, be the set of blocks after dividing sequence $\mathcal{V}$ by every h^* messages. We use $R(\mathcal{X})$ and $H(\mathcal{X})$ to represent the number of unique recipients and the entropy of $\mathcal{X}$ ($\mathcal{X}$ can be a block $\mathcal{B}_i$ or the test sequence $\mathcal{T}$).

We let $\mathbb{E}(R_\mathcal{B})$ and $var(R_\mathcal{B})$ denote the mean and variance of $\{R(\mathcal{B}_i) : 1 \leq i \leq l\}$, and $\mathbb{E}(H_\mathcal{B})$ and $var(H_\mathcal{V})$ denote the mean and variance of $\{H(\mathcal{B}_i) : 1 \leq i \leq l\}$.

We perform mean-based anomaly detection as follows: if $|R(\mathcal{T}) - \mathbb{E}(R_\mathcal{B})| > \alpha_R$, we raise an *R-type* alert (**R-type detection**); if $|H(\mathcal{T}) - \mathbb{E}(H_\mathcal{B})| > \alpha_H$, we raise an *H-type* alert (**H-type detection**). The rationale behind it is simple: if the mean observed from the test sequence deviates from the mean observed from the training trace by a predefined threshold, we deem it as anomalous.

Then, an important question is how to choose thresholds α_R and α_H. A too large threshold may miss many anomalous behaviors but a too low threshold may raise too many false alerts. We do this based on the *Chebyshev's inequality*:

$$\mathbb{P}\{|R(\mathcal{T}) - \mathbb{E}(R_\mathcal{B})| > \alpha_R\} \leq \frac{var(R_\mathcal{B})}{\alpha_R^2}. \quad (4)$$

Let β_R be the upper bound on the expected false alarm rate for R-type alerts. In practice, β_R is a configurable input parameter. By having $\alpha_R = \sqrt{var(R_\mathcal{B})/\beta_R}$, we ensure that the expected false alarm rate of R-type alerts does not exceed β_R. Similarly, choosing $\alpha_H = \sqrt{var(H_\mathcal{B})/\beta_H}$, where β_H gives the upper bound on the expected false alarm rate for H-type alerts, renders the expected false alarm rate of H-type alerts no greater than β_H.

From an **implementation** point of view, the mean-based anomaly detection scheme imposes trivial computational overhead. For each SMS user, it only requires only four states for anomaly detection: $\mathbb{E}(R_\mathcal{B})$, $\mathbb{E}(H_\mathcal{B})$, α_R, and α_H. The parameters can be derived from the training trace in an offline fashion, but their values can be stored in the memory (instead of on disk), thereby relieving the online anomaly detection from intensive disk access operations.

Similarity-based anomaly detection. We now explore how to exploit similarity-based metrics for anomaly detection. A naive implementation can be the following: we compute a similarity metric (recipient similarity metric or JS-divergence) between the test sequence $\mathcal{T}$ and each block in the history trace and check whether its mean significantly deviates from the mean similarity metric only between the blocks in the history trace. Although straightforward, this scheme demands knowledge of the whole history trace when performing online anomaly detection, thereby rendering it hardly practical due to its prohibitive computational cost.

Due to such performance concern, we propose a light-weight anomaly detection scheme as follows. First, instead of comparing the test sequence $\mathcal{T}$ against *each block* in the history trace, we condense information in the history trace into a set of recipients and a distributional function. Furthermore, we do not consider the entire set of recipients that have been witnessed in the history trace, but instead focus on the top few recipients that have received the most messages from the SMS user. Such simplification is justified by the previous results showing that the similarity metrics bear low variation even if only the top few recipients are considered within each message block.

Suppose that we only consider the top ϕ recipients. Let $\mathcal{G}_\phi(X)$ denote the set of the top ϕ recipients that receive the most messages within sequence X, and $\mathcal{Q}_\phi(X)$ the normalized distribution of the number of short messages sent

to the top ϕ recipients within sequence X. The similarity-based anomaly detection scheme checks how significantly $S(\mathcal{G}_\phi(\mathcal{T}), \mathcal{G}_\phi(\mathcal{V}))$ and $D_{\text{JS}}(\mathcal{Q}_\phi(\mathcal{T}) \| \mathcal{Q}_\phi(\mathcal{V}))$ deviate from the means that have been observed from the history trace.

Recall that $\{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_l\}$ is the set of blocks after dividing sequence $\mathcal{V}$ by every h^* messages. To compute the means, we first define $\mathcal{V} \setminus \mathcal{B}_i$, where $1 \leq i \leq l$, as the sequence after block B_i is removed from $\mathcal{V}$. We then let $\mathbb{E}(S_\phi)$ and $var(S_\phi)$ be the mean and variance of elements in set $\{S(\mathcal{G}_\phi(\mathcal{B}_i), \mathcal{G}_\phi(\mathcal{V} \setminus \mathcal{B}_i)) : 1 \leq i \leq l\}$. Similarly, we use $\mathbb{E}(D_\phi)$ and $var(D_\phi)$ to denote the mean and variance of elements in set $\{D_{\text{JS}}(\mathcal{Q}_\phi(\mathcal{B}_i) \| \mathcal{Q}_\phi(\mathcal{V} \setminus \mathcal{B}_i)) : 1 \leq i \leq l\}$. Given $\mathcal{V}$ and h^*, we can easily calculate $\mathbb{E}(S_\phi)$, $var(S_\phi)$, $\mathbb{E}(D_\phi)$ and $var(D_\phi)$.

Similarity-based anomaly detection on test sequence $\mathcal{T}$ works as follows: if $|S(\mathcal{G}_\phi(\mathcal{T}), \mathcal{G}_\phi(\mathcal{V})) - \mathbb{E}(S_\phi)| > \alpha_S$, we raise an *S-type* alert (**S-type detection**); if $|D_{\text{JS}}(\mathcal{Q}_\phi(\mathcal{T}) \| \mathcal{Q}_\phi(\mathcal{V})) - \mathbb{E}(D_\phi)| > \alpha_D$, we raise a *D-type* alert (**D-type detection**). Using the Chebyshev's inequality, we can determine parameters α_S and α_D as follows: $\alpha_S = \sqrt{var(S_\phi)/\beta_S}$ and $\alpha_D = \sqrt{var(D_\phi)/\beta_D}$, where β_S and β_D are the upper bounds on the expected false alarm rates for S-type and D-type alerts, respectively. Both β_S and β_D are input parameters in practice.

From the **implementation** perspective, the similarity-based anomaly detection schemes do not impose high computational cost. For each SMS user, we can compute the four variables $\mathbb{E}(S_\phi)$, $var(S_\phi)$, $\mathbb{E}(D_\phi)$ and $var(D_\phi)$ based on her history trace $\mathcal{V}$ in an offline fashion. We then calculate α_S and α_D accordingly. When we perform online anomaly detection, we need to know not only $\mathbb{E}(S_\phi)$, $\mathbb{E}(D_\phi)$, α_S and α_D, but also $\mathcal{G}_\phi(\mathcal{V})$ and $\mathcal{Q}_\phi(\mathcal{V})$. Clearly, the sizes of $\mathcal{G}_\phi(\mathcal{V})$ and $\mathcal{Q}_\phi(\mathcal{V})$ depend on ϕ. In total, the S-type detection requires at most $\phi + 2$ states and the D-type detection requires at most $2\phi + 2$ states. Since ϕ is usually much smaller than the size of the set of unique recipients shown in history trace $\mathcal{V}$, the computational cost of our proposed scheme improves significantly compared to the aforementioned naive similarity-based scheme that demands the full knowledge of the whole history trace. Our experiments later show that $\phi = 5$ is sufficient to achieve high detection accuracy. In that case, even for the D-type detection scheme, only 12 states are needed.

6 Experimental Evaluation

Setup. In this section, we shall evaluate the performance of our proposed detection schemes. For this, we use the same data trace discussed in Section 3. As our detection schemes require some training data to derive a few parameters, we consider only those SMS users that have sent out at least 200 short messages. In total, there are 167 such users. Here note that the discarded user data are not relevant because of the limited SMS traffic they produce and are thus not a concern. We also use 70% of each SMS user's short messages for training and the remaining 30% for testing. Suppose that the number of training short messages is n. We let h_{min} be 10 and h_{max} be $\min\{30, \lfloor n/10 \rfloor\}$ in Equation (4). We believe that bounding h between 10 and 30 provides a good balance between detection accuracy and latency. Further, we also bound h from the upper side by

$\lfloor n/10 \rfloor$ to ensure that there are at least 10 elements for variance computation; if we have enough training data, such a constraint can be relieved. We also have: $\beta_R = \beta_H = \beta_S = \beta_D = \beta$, and vary β between 0.05 and 0.1.

False positive rates. The false positive rates of the four detection schemes are shown in the following table:

Scheme	$\beta = 0.05$	$\beta = 0.1$
R-type detection	1.0%	2.2%
H-type detection	0.8%	2.7%
S-type detection	0.0%	5.4%
D-type detection	0.0%	4.3%

From the above table, we observe that the false positive rates of all four detections are very low. The effect of β on the false positive rates is also obvious: a higher β leads to a higher false positive rate, irrespective of the type of alerts considered. This is because a higher β lowers threshold α_R in Equation (4) (or α_H, α_S, and α_D in the other three cases).

Detection rates of blending attacks. In our experiments, we consider every pair of SMS users in which one is the victim and the other is the attacker. Suppose that SMS user a is the victim and b is the attacker. We first identify the timestamp of the last message in user a's training dataset; we further get the list of messages that are sent by user b that are sent *after* that timestamp in the trace. For brevity, we call this list an *attack list*. Since it is possible that the attack list may not have enough messages, e.g., because user b quit from system before data collection terminated, we only consider those cases that have at least $4h_a^*$ messages on the attack list, where h_a^* is the detection window size for user a. Messages on the attack list are then merged with those in user a's test dataset, with their timestamp ordering unchanged as in the original trace.

Among all pairs of SMS users considered, we compute the fraction of cases in which the blending attack is successfully detected by each scheme, and the average detection delay in the number of detection windows if the attack is indeed detected. The results are as follows:

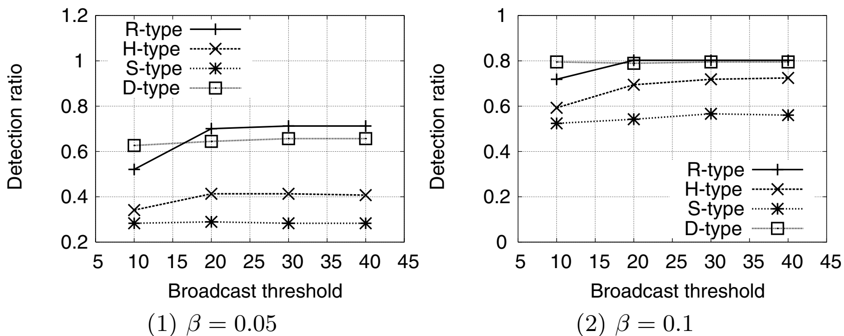
Scheme	$\beta = 0.05$		$\beta = 0.1$	
	Rate	Delay	Rate	Delay
R-type	35.6%	3.9	55.6%	4.2
H-type	40.5%	1.8	62.3%	2.8
S-type	44.1%	1.0	71.6%	1.0
D-type	65.1%	1.0	81.7%	1.0

Three observations can made from the above table. *First*, similarity-based schemes can detect blending attacks with higher rates and smaller detection delays than mean-based schemes. This is because similarity-based schemes encode more information in the detection metrics. *Second*, both H-type detection

and D-type detection consider not only the set of unique recipients, but also the distribution of the number of short messages sent to each recipients. Hence, they perform better than both the R-type and S-type detection schemes. *Third*, a higher β leads to a higher detection threshold, thereby improving both the detection rate and the detection delay, irrespective of the detection scheme.

Detection rates of broadcast attacks. In the experiments, we intermingle the test dataset of each SMS user with malicious messages sent to recipients that are *randomly* chosen from those observed in the training dataset. For each SMS user, exactly γ malicious messages are sent out at 12:00PM every day. We call γ the *broadcast threshold*, which is varied among 10, 20, 30, and 40.

The detection ratios are depicted in Figure 13. Unsurprisingly, detection ratios when $\beta = 0.1$ are higher than those when $\beta = 0.05$. We note, however, that the relative ranks of the four schemes differ significantly from the detection results for blending attacks. Detection based on recipient similarity metric (i.e., S-type detection) performs the worst but detection simply based on the number of unique recipients (i.e., R-type detection) performs quite well. Recall that R-type detection for blending attacks is not as effective as the other three schemes. Such difference actually attributes to the type of attacks we are considering. For broadcasting attacks, as recipients of illegitimate short messages are actually drawn from those recipients of those legitimate short messages, the change on the recipient similarity metric under broadcasting attacks is limited. Broadcast attacks, however, generate a large number of messages with different recipients, thereby exposing themselves to the R-type detection scheme which simply monitors the change on the number of unique recipients within each detection window. On the other hand, Figure 13 reveals that D-type detection is still effective against broadcast attacks. This is because although broadcast attacks mimic the set of recipients that have been observed in the training dataset, the distribution of the number of messages sent to each recipient is still different from that in the training dataset.



(1) $\beta = 0.05$ (2) $\beta = 0.1$

Fig. 13. Detection ratio of broadcast attacks

The average detection delay in the number of detection windows are given in the following table:

β	R-type	H-type	S-type	D-type
0.05	4.0	3.7	6.8	4.1
0.1	3.2	3.0	5.5	3.2

Recall that on average similarity-based schemes detect blending attacks within a single detection window (if the detection is successful). For broadcast attacks, however, detection delays are higher because illegitimate short messages are sent at the same time in a day in our experiments and the detector thus has to wait for that moment to catch these attacks.

Hybrid detection. We now explore the detection solution space further by combining multiple detection schemes together. Due to space limitation, we consider only two hybrid detection schemes: In the first one (**R/H/S/D**), if any type of alert is flagged, we treat it as anomalous; otherwise, we treat it as normal; in the second one (**S/D**), if an S- or D-type of alert is flagged, we treat it as anomalous; otherwise, we treat it as normal. The following table provides the performance of these two schemes ($DR_{blending}$ and $DR_{broadcast}$ denote the detection ratio of blending and broadcast attacks, respectively):

	R/H/S/D		**S/D**	
	$\beta = 0.05$	$\beta = 0.1$	$\beta = 0.05$	$\beta = 0.1$
False alarm rate	1.3%	8.5%	0.0%	5.0%
$DR_{blending}$	85.7%	96.2%	69.1%	83.4%
$DR_{broadcast}$ ($\gamma = 10$)	78.3%	94.0%	65.7%	82.5%
$DR_{broadcast}$ ($\gamma = 20$)	82.5%	92.8%	68.1%	80.7%
$DR_{broadcast}$ ($\gamma = 30$)	83.7%	93.4%	69.3%	81.9%
$DR_{broadcast}$ ($\gamma = 40$)	83.1%	93.4%	69.3%	81.9%

We note that the **R/H/S/D** scheme with $\beta = 0.1$ can catch blending and broadcast attacks with detection ratios higher than 90% but at the expense of a relatively high false positive rate, which is about 8.5%; when $\beta = 0.05$, the false alarm rate is only 1.3% but its detection ratios of blending and broadcast attacks fall between 78% and 86%. The **S/D** scheme, although not able to detect as many attacks as the **R/H/S/D** scheme with the same β, does not generate any false alarm when $\beta = 0.05$, and still catches about two thirds of the attacks.

There is a clear tradeoff between high detection rates and low false alarm rates. In practice, the decision on which parameterized detection scheme to use can be made based on the user's preference on this tradeoff, which is also affected by the frequency at which she needs to deal with a false alarm. Here, we provide simple analysis on the average interval between two false alarms. Suppose that each detection window contains h short messages and the false alarm rate is p. By modeling false alerts as a Bernoulli process, the average number of windows before a false alarm is raised is $1/p$. Hence, the average number of

messages between two false alarms is h/p. Consider the case with $h = 20$ and $p = 8\%$. Note that we are considering a relatively high false alarm rate. Then, about every 250 short messages leads to a false alarm. In our trace, we observe that a persistent user sends 1.5 messages per day on average, suggesting that a normal SMS user needs more than 5 months on average to receive a false alarm. Even if we consider the largest average daily number of short messages sent by an SMS user in our trace, which is about 25, a false alarm is raised every 10 days.

7 Conclusions and Future Work

The goal of this work is to detect anomalous SMS behaviors. From an SMS trace that was collected within a five-month period, we observe that there are window-based regularities inherent in behaviors of typical SMS users. Accordingly, we develop SMS-Watchdog, a light-weight detection scheme that relies on normal social behavior profiles built for each SMS user. Experimental results show that our detection approach can detect more than 92% of SMS-based attacks with false alarm rate 8.5%, or about two thirds of the attacks without any false alarm.

Admittedly, SMS-Watchdog is not panacea for all SMS-related attacks. For instance, SMS-Watchdog is not able to detect SMS faking attacks, as such attacks simulate the behavior of SMS switches (i.e., SMSC in Figure 1) and the illegitimate SMS messages do not go through the SMSC of the originating terminal, where the SMS-Watchdog is deployed. Moreover, with the integration of the telecommunication network and the Internet, many SMS messages are now sent from the Internet. SMS accounts can be easily created through the Internet and then used to send spamming or phishing SMS messages. Given the fact that SMS-Watchdog requires a training process to build a behavioral profile for each SMS user, it is difficult for SMS-Watchdog to identify those transient SMS accounts that are used only for spamming or phishing purposes.

Moreover, as SMS-Watchdog detects abnormal SMS activities by monitoring deviations from behavioral profiles trained under normal circumstances, it is possible that some malware can intelligently evade its detection. For example, a stealthy malware can learn the behavior of an SMS user from her recent SMS communication history and then send spamming or phishing SMS messages in a similar fashion. Also, the design of SMS-Watchdog takes the variation of a typical SMS user's regular behavior into consideration to avoid high false positive rates. Accordingly, a stealthy malware can exploit this to evade its detection by limiting the number of illegitimate SMS messages sent within each detection window.

With the lesson learned from the increasing sophistication of cyber-attacks in the Internet, we do not claim that SMS-Watchdog can address all existing or future SMS-related attacks. While we will continuously improve the effectiveness of SMS-Watchdog against these attacks, we also plan to explore other complementary approaches to protect this increasingly popular service.

References

1. Bose, A., Hu, X., Shin, K.G., Park, T.: Behavioral detection of malware on mobile handsets. In: Proceedings of MobiSys 2008 (2008)
2. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM Computing Survey (September 2009) (to appear)
3. Cover, T., Thomas, J.: Elements of Information Theory. John Wiley, Chichester (1991)
4. http://www.redherring.com/Home/19081
5. Davis, A.B., Goyal, S.K.: Knowledge-based management of cellular clone fraud. In: Proceedings of IEEE PIMRC 1992, Boston, MA, USA (1992)
6. Enck, W., Traynor, P., McDaniel, P., Porta, T.L.: Exploiting open functionality in SMS-capable cellular networks. In: Proceedings of CCS 2005 (2005)
7. Fawcett, T., Provost, F.: Activity monitoring: noticing interesting changes in behavior. In: Proceedings of ACM KDD 1999 (1999)
8. Hu, G., Venugopal, D.: A malware signature extraction and detection method applied to mobile networks. In: Proceedings of IPCCC 2007 (April 2007)
9. Kim, H., Smith, J., Shin, K.G.: Detecting energy-greedy anomalies and mobile malware variants. In: Proceedings of MobiSys (2008)
10. Lee, L.: Measures of distributional similarity. In: Proceedings of the 37th Annual Meeting of the ACL (1999)
11. Lee, W., Xiang, D.: Information-theoretic measures for anomaly detection. In: Proceedings of IEEE S&P (2001)
12. Lin, Y., Chlamtac, I.: Wireless and Mobile Network Architectures. John Wiley & Sons, Inc., Chichester (2001)
13. Meng, X., Zerfos, P., Samanta, V., Wong, S.H.Y., Lu, S.: Analysis of the reliability of a nationwide short message service. In: Proceedings of INFOCOM 2007 (2007)
14. Noble, C.C., Cook, D.J.: Graph-based anomaly detection. In: KDD 2003: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (2003)
15. http://www.vnunet.com/vnunet/news/2163586/sms-phishing-attack-seen-wild
16. http://www.kval.com/news/local/17945949.html
17. http://www.mobilemarketer.com/cms/opinion/columns/1610.html
18. http://www.textually.org/textually/archives/2007/12/018482.htm
19. http://www.openmindnetworks.com/SMSSpoofing.asp
20. Stolfo, S.J., Hershkop, S., Hu, C., Li, W., Nimeskern, O., Wang, K.: Behavior-based modeling and its application to email analysis. ACM Transactions on Internet Technology 6(2), 187–221 (2006)
21. Sun, B., Yu, F., Wu, K., Xiao, Y., Leung, V.C.M.: Enhancing security using mobility-based anomaly detection in cellular mobile networks. IEEE Trans. on Vehicular Technology 55(3) (2006)
22. http://searchcio-midmarket.techtarget.com/tip/0,289483,sid183_gci1310706,00.html
23. Taniguchi, M., Haft, M., Hollmn, J., Tresp, V.: Fraud detection in communications networks using neural and probabilistic methods. In: Proceedings of the 1998 IEEE International Conference in Acoustics, Speech and Signal Processing (1998)
24. Traynor, P., Enck, W., McDaniel, P., Porta, T.L.: Mitigating attacks on open functionality in SMS-capable cellular networks. In: Proceedings of MobiCom 2006 (2006)

25. Yan, G., Eidenbenz, S., Sun, B.: Mobi-watchdog: you can steal, but you can't run! In: Proceedings of ACM WiSec 2009, Zurich, Switzerland (2009)
26. Yan, G., Xiao, Z., Eidenbenz, S.: Catching instant messaging worms with change-point detection techniques. In: LEET 2008: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats, Berkeley, CA, USA (2008)
27. Zerfos, P., Meng, X., Samanta, V., Wong, S.H.Y., Lu, S.: A study of the short message service of a nationwide cellular carrier. In: Proceedings of IMC 2006 (2006)

Keystroke-Based User Identification on Smart Phones

Saira Zahid[1], Muhammad Shahzad[1], Syed Ali Khayam[1,2], and Muddassar Farooq[1]

[1] Next Generation Intelligent Networks Research Center (nexGIN RC)
National University of Computer & Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
{saira.zahid,muhammad.shahzad,muddassar.farooq}@nexginrc.org
[2] School of Electrical Engineering & Computer Science (SEECS)
National University of Sciences & Technology (NUST)
Islamabad, Pakistan
ali.khayam@seecs.edu.pk

Abstract. Smart phones are now being used to store users' identities and sensitive information/data. Therefore, it is important to authenticate legitimate users of a smart phone and to block imposters. In this paper, we demonstrate that keystroke dynamics of a smart phone user can be translated into a viable features' set for accurate user identification. To this end, we collect and analyze keystroke data of 25 diverse smart phone users. Based on this analysis, we select six distinguishing keystroke features that can be used for user identification. We show that these keystroke features for different users are diffused and therefore a fuzzy classifier is well-suited to cluster and classify them. We then optimize the front-end fuzzy classifier using Particle Swarm Optimization (PSO) and Genetic Algorithm (GA) as back-end dynamic optimizers to adapt to variations in usage patterns. Finally, we provide a novel keystroke dynamics based PIN (Personal Identification Number) verification mode to ensure information security on smart phones. The results of our experiments show that the proposed user identification system has an average error rate of 2% after the detection mode and the error rate of rejecting legitimate users drops to zero in the PIN verification mode. We also compare error rates (in terms of detecting both legitimate users and imposters) of our proposed classifier with 5 existing state-of-the-art techniques for user identification on desktop computers. Our results show that the proposed technique consistently and considerably outperforms existing schemes.

1 Introduction

Smart phones[1] are pervasively and ubiquitously integrating into our home and work environments. In particular, due to the enhanced capabilities available on contemporary smart phones, users – in addition to personal and professional

[1] We use the terms smart phone and mobile phone interchangeably throughout this paper because our proposed system, with the support of OS vendors, can be deployed on both types of phones.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 224–243, 2009.
© Springer-Verlag Berlin Heidelberg 2009

contacts' information – now store sensitive information such as emails, credit card numbers, passwords, corporate secrets, etc. on mobile phones which make them an attractive target for imposters [9]. Stolen mobile phones can be used for identity theft which can then be exploited for malicious and/or unlawful activities. Various surveys conducted recently show that in case of mobile phone theft, users (instead of being worried about the cost of the stolen phone) are becoming more concerned with the misuse of information and services on the stolen phones [1]. Therefore, it is important to develop intelligent user identification schemes for mobile phones.

Despite its need and importance, user identification on mobile phones has received little attention in research literature. User identification systems for mobile phones are usually based on secret PIN numbers [28]. These identification techniques are borrowed from desktop computers' domain and have not been very effective on mobile phones [6],[28]. For instance, freely available tools empower intruders, who have physical access to the Subscriber's Identity Module (SIM) and know the Personal Identification Number (PIN), to reverse engineer the International Mobile Subscriber Identity (IMSI) and the secret key of GSM mobile phone users [19]. Similarly, token-based authentication schemes developed for desktops are not suitable for mobile phones because: (1) they cannot be efficiently implemented on resource-constrained devices [9], and (2) loss of a token in essence means loss of the device [26]. Biometric hardware for mobile phones are now being developed to overcome the shortcomings of token-based authentication [26]. A common drawback of these authentication paradigms is that they perform one-time identity check at the beginning of a session that allows imposters to access the smart phones once a session has been logged in.

In this paper, we propose a robust approach to identify a legitimate user of a mobile phone by learning his/her "in-session" keystroke dynamics. The scheme requires no additional hardware or software resources and is user-friendly as it requires minimum user intervention after installation. While keystroke-based user identification was actively pursued in the domain of desktop computer [27],[17],[14],[18], its suitability for mobile phones has not been explored, except by the preliminary work reported in [7],[15]. To use keystroke information for user identification, we collect and analyze keystroke data of 25 diverse mobile phone users including researchers, students, and professionals from varying age groups. Based on our analysis, we select six distinguishing keystroke features that can be used for user identification. Two of these features – *key hold time* (how long a key is pressed) and *error rate* (number of times backspace is pressed) – are borrowed from the desktop domain. We also customize a set of four features to capture the unique switching behavior across multiplexed mobile phone keys[2] using: (1) *Horizontal Digraph*: time to switch between horizontally adjacent keys, (2) *Vertical Digraph*: time to switch between vertically adjacent

[2] This study has been done only for the smart phones with numeric keypads. In these phones, each key stands for multiple characters that can be produced by pressing the key a predefined number of times. We thus name the keys of such phones as multiplexed keys.

keys, (3) *Non-Adjacent Horizontal Digraph*: time to switch between non-adjacent horizontal keys, and (4) *Non-Adjacent Vertical Digraph*: time to switch between non-adjacent vertical keys.

We reveal that, while these keystroke features differ across users, leveraging them for accurate user identification on mobile phones is significantly more challenging than on a desktop computer because on a majority of contemporary mobile phones: (1) different keys are multiplexed on a small keypad, (2) the variable and discontinuous keystroke usage of a mobile phone user results in a highly diffused (overlapping) and time-varying feature space that makes it difficult to cluster and classify different users, and (3) an imposter can get access to a mobile phone at anytime so techniques that rely on static, application-specific or keyword-specific authentication are not feasible. These challenges are aggravated by the fact that most of the mobile OS vendors do not provide any mechanism for key logging. In view of these challenges, we set two accuracy objectives for the proposed technique: (1) correctly identify imposters and legitimate users using keystroke dynamics[3], and (2) identify an imposter within a small number of key hits to ensure timely information security. In addition to being accurate, an effective user authentication scheme for mobile phones must: (1) be able to continuously adapt to varying usage patterns of a phone user, (2) utilize a classifier that provides high classification accuracy for a diffused features space, and (3) have low-complexity so that it can be deployed on resource-constrained mobile phones.

To meet the above requirements, we propose a keystroke-based user identification system which operates in three sequential modes.

Learning Mode. In this mode, we train a fuzzy classifier which maps the diffused feature space of a mobile phone user to his/her profile. Moreover, it utilizes a hybrid of bio-inspired optimizers – Particle Swarm Optimization (PSO) [16] and Genetic Algorithm (GA) [11] – at the back-end for continuous evolution of the fuzzy system in order to cope with the varying usage pattern of the user[4].

Imposter Detection Mode. In this mode, the trained classifier is used to classify real-time keystroke measurements to classify a user as legitimate or imposter.

Verification Mode. This mode is only invoked if a user is potentially identified as an imposter in the detection mode or the user wants to transmit documents from a mobile phone. In the verification mode, the potential imposter is asked to type a memorized 8-character PIN of the legitimate user. The system then uses the keystroke dynamics model to analyze the typing behavior of the potential imposter. This mode makes it difficult for an imposter to have illegitimate access

[3] Throughout the paper we define accuracy in terms of error in detecting an imposter – False Acceptance Rate (FAR), and error in detecting a legitimate user – False Rejection Rate (FRR).

[4] PSO and GAs are well-known for providing efficient and online solutions to dynamic and time-varying optimization problems [10],[4].

by hacking the PIN only; therefore, it serves as the last line of defence after an imposter has breached all other security layers.

Performance evaluation on the collected dataset shows that the proposed hybrid PSO-GA based fuzzy classifier, when trained using a mere 250 keystrokes, achieves an average error rate of approximately 2% after the detection mode and an FRR close to zero after the verification mode. We compare the accuracy of our system with five other state-of-the-art keystroke-based user identification techniques and show that our proposed system provides significantly better accuracy in detecting legitimate users and imposters.

The rest of the paper is organized as follows. Section 2 briefly describes the related work. We discuss our dataset in Section 3 and explain feature selection along with a study of existing desktop schemes on these features in Section 4. In Section 5, we first investigate the feasibility of existing desktop classification schemes for mobile phones, and then we discuss the architecture of our proposed user identification system. In Section 6, we analyze the performance of our proposed system for varying parameters. The limitations of the proposed system and potential countermeasures to overcome these limitations are detailed in Section 7. Finally, we conclude the paper with an outlook to our future research.

2 Related Work

The idea of using keystroke dynamics for user authentication is not new as there have been a number of prior studies in this area for desktop computers. Most of these studies have focused on static or context-independent dynamic analysis using the inter-keystroke latency method for desktop keyboards only. From the earliest studies in 1980 [5], the focus has been on the analysis of delay between two consecutive keystrokes – also called digraph. Later studies [14],[20] further enhanced the work by identifying additional statistical analysis methods that provided more reliable results. This section briefly summarizes some of the prominent research on keystroke based user identification.

One of the earlier works in the area of keystroke dynamics was accomplished by Umphress and Williams [27] in 1985. They used digraphs as the underlying keystroke biometric. However, they were only able to achieve an FAR of 6%. In 1987, Williams and Leggett [17] further extended the work by: (1) increasing the number of users in the study, (2) reducing experimental variables, and (3) discarding inappropriate digraphs according to latency and frequency. They managed to reduce the FAR to 5%.

Another extension of the above work was conducted in 1990 by Leggett et al. [18]. While the results of the static procedure of entering a reference and testing profiles achieved the same 5% FAR, they were the first ones to utilize the concept of keystroke dynamics for doing verification in a dynamic environment. They were able to achieve FAR of 12.8% and FRR of 11.1% using statistical theory. In a study by Joyce and Gupta [14], the username was compared to the particular profile for that user. The login had four components – username, password, first name, and last name. Digraphs were then calculated and basic

statistical method of means, variances, and standard deviations were used to determine a match. Using this method, the FAR was just 0.25% but the FRR was 16.67%. Bleha et al. [3], in 1990, used a different statistical method: the Bayes classification algorithm. The verification system gave results of 8.1% for FRR and 2.8% for the FAR. Regarding features' set, no significant additions occurred until 1997 when Obaidat and Sadoun [21] introduced key hold times as another feature of interest. Currently, the most common and widely-known application that uses keystroke dynamics technology is BioPassword [12]. To the best of our knowledge, BioPassword is the only product available in the market that has relatively wide usage.

These studies, however, have focused their research only on desktop computers. Except for [7],[15], no work has been done on user identification using keystroke dynamics on mobile phones. Clarke et al. [7] have used neural networks to classify a user by using key hold time and inter-key latency. They performed three sets of experiments on mobile phone emulators: (1) on PIN verification, (2) on specific text, and (3) on phone number entry. They achieved FARs of 3%, 15% and 18% respectively for these three experiments, however FRRs were 40%, 28% and 29%, respectively.

3 Data Acquisition

As a first step towards developing a robust mobile phone user identification system, we developed an application to log mobile keystroke data. We decided to develop the application for Symbian OS 3rd Edition because: (1) it had a relatively large customer base in our social network, and (2) it provides developers with Application Programming Interfaces (APIs) to capture key events. The application runs in the background so that a user can continue using his/her mobile phone uninterruptedly. All keys pressed by a user are logged along with the press/release times of the keys[5]. In addition to the regular keys, we also log left soft key, right soft key, left arrow key, right arrow key, up arrow key, down arrow key, joystick key, menu key, call dial key, call end key, back space key, camera key, volume up key, volume down key, * key, and # key. A text file containing all logged key events is stored in the phone memory and is periodically uploaded to our development server. The application was digitally signed by Symbian Signed (`http://www.symbiansigned.com`) before deployment.

Despite security and privacy concerns shown by most volunteers, we were able to convince 25 mobile phone users to volunteer for this study. The subjects of our study have different socioeconomic backgrounds (see Table 1) that provides good diversity in our dataset; we have teenagers, corporate executives, researchers, students, software developers and even a senior citizen in our list of volunteers.

Another distinguishing feature of this dataset is that it is not a one prototype model dataset and has been collected from a diverse set of Nokia mobile phones. We have N-series, E-series and 6xxx series mobile phones all of which have multiplexed keypad. This diversity in phone sets is important to ensure that

[5] We used Active Objects to realize this functionality [2].

Table 1. Feature table of 25 mobile phone users of this study (c_v is coefficient of variation)

Users	Nokia model	Social status	Total key hit profiles	Total key hits	Key Hold Time μ (ms)	c_v	Horizontal Digraph μ (sec)	c_v	Vertical Digraph μ (sec)	c_v	Non-adjacent Vertical Digraph μ (sec)	c_v	Non-adjacent Horizontal Digraph μ (sec)	c_v	Error (%)
u1	N73	manager	17	4113	61.1	0.08	0.12	2.08	0.15	1.80	0.19	1.21	0.05	1.11	1.25
u2	N95	researcher	48	11901	83.1	0.01	0.32	0.87	0.33	0.87	0.24	1.79	0.09	1.23	3.07
u3	N81	student	12	2939	81.4	0.04	0.35	0.31	0.42	0.45	0.39	1.43	0.07	1.36	2.63
u4	6650	engineer	21	5011	131	0.02	0.26	0.88	0.34	0.38	0.18	1.77	0.13	0.82	0.93
u5	6120	teenager	34	8255	103	0.08	0.21	2.09	0.12	2.33	0.43	1.25	0.11	1.31	2.38
u6	N79	businessman	20	4919	25.3	0.16	0.32	1.68	0.17	3.05	0.53	0.45	0.14	1.23	8.47
u7	N73	student	30	7283	45.1	0.21	0.11	5.72	0.23	2.82	0.12	5.33	0.17	1.11	8.59
u8	6124	manager	33	8209	95.9	0.03	0.12	1.50	0.11	3.09	0.53	0.43	0.09	0.79	5.79
u9	N95	engineer	37	9211	83.2	0.04	0.31	2.45	0.61	0.39	0.32	1.03	0.05	1.09	4.10
u10	N82	advertiser	53	13193	76.6	0.04	0.21	0.62	0.42	0.59	0.52	1.46	0.08	1.23	6.55
u11	E51	student	27	6501	32.1	0.12	0.33	0.87	0.56	1.33	0.32	0.68	0.04	1.01	8.14
u12	6120	researcher	37	9028	67.3	0.03	0.18	1.66	0.82	0.42	0.54	0.62	0.11	0.92	7.85
u13	N81	student	41	10001	11.3	0.16	0.22	1.59	0.28	0.82	0.75	0.88	0.14	1.22	6.02
u14	N77	student	53	13022	35.6	0.07	0.24	1.37	0.48	1.56	0.35	1.22	0.19	0.86	2.47
u15	E65	engineer	55	13713	61.5	0.05	0.33	0.93	0.41	0.95	0.12	1.83	0.16	1.31	3.22
u16	N76	senior citizen	19	4744	15.9	0.13	0.71	0.33	0.86	0.75	0.43	1.55	0.13	0.87	2.37
u17	N81	manager	12	2900	42.1	0.07	0.54	1.40	0.28	1.25	0.65	0.35	0.08	2.01	11.2
u18	6121	engineer	48	11793	57.6	0.07	0.18	1.72	0.48	1.56	0.45	1.20	0.13	1.71	1.35
u19	6120	student	17	4011	21.7	0.01	0.21	3.38	0.19	2.94	0.19	1.21	0.07	1.52	1.21
u20	N73	researcher	47	11529	33.4	0.15	0.15	1.53	0.32	0.81	0.43	1.23	0.15	1.08	2.53
u21	N81	researcher	20	4992	76.3	0.02	0.66	0.51	0.64	0.67	0.64	0.35	0.15	0.57	1.33
u22	N73	director	27	6721	23.3	0.12	0.21	1.04	0.24	3.25	0.24	2.79	0.11	1.14	6.23
u23	N95	student	41	10132	68.2	0.08	0.63	0.61	0.53	0.66	0.15	2.26	0.14	1.16	8.43
u24	N81	researcher	39	9531	79.7	0.05	0.44	0.72	0.31	0.74	0.32	1.71	0.09	1.11	3.11
u25	6120	teenager	33	8193	17.5	0.38	0.25	0.64	0.45	1.66	0.74	0.31	0.07	1.23	2.31

the design of our system and its evaluation spans across a wide range of modern mobile phones. The complete dataset is available at `http://www.nexginrc.org`.

For all the analysis provided later in the paper, we use a dataset of 25 users spanning over 7 days. We quantify the keystrokes into a profile of 250 key-hits[6] each, which we call a 'Key hit profile'. Table 1 shows that people from different walks of life have different number of key hit profiles in accordance with their social status. We observe that students, teenagers and professionals use keyboard of mobile phones aggressively while senior citizens and managers use keyboard of mobile phone less frequently. For instance, users u10, u14, and u15 have more than 50 key hit profiles while users u1, u3, u16, u17, and u19 make less than 20 key hit profiles over the same period of 7 days.

After successfully collecting the dataset, we started the next phase of our research – systematically analyzing our raw data to extract useful features for user identification. We observed that some people tend to type faster with less errors as compared to others, while some others type very slowly which is uniquely linked to their social status and age as shown in Table 1. Based on this preliminary analysis, we observed that if we can identify a keystroke dynamics feature set that covers all aspects of a persons' unique typing pattern, we can actually identify the mobile phone user. Therefore, we extracted 6 features to correctly identify a user – 2 of these features have been borrowed from the desktop domain while the remaining 4 are customized for mobile phones' multiplexed keypads. A detailed discussion of this features' set is provided in the next section.

[6] A justification for this profile size is given during the discussion of experiments.

4 Feature Selection and Study of Desktop-Based Schemes

In this section, we first analyze three well-known features that have been used for user identification on desktop/laptop computers. We then customize these features for mobile phones. Finally, we evaluate the accuracies of existing keystroke-based user identification schemes in identifying mobile phone users.

4.1 Feature Selection

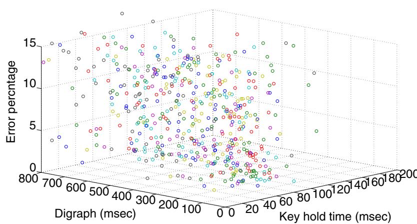
After collecting data of the mobile phone users, we extracted three features from this data – key hold time, digraph, and error rate. These features have been used for keystroke-based user identification for desktop/laptop computers [17],[14]. However, their usability to identify a legitimate user on mobile phones has not been explored before. These features are defined as:

Key hold time. The time difference between pressing a key and releasing it;

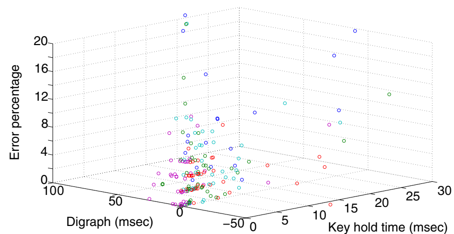
Digraph time. The time difference between releasing one key and pressing the next one; and

Error rate. The number of times backspace key is pressed.

We observed that identifying a user based on these three features is less challenging on desktops because of a relatively distinguished feature vector for each user. As an example, we installed a key-logging application on the laptops of 6 users for a period of 5 days. The plot of these three features extracted from the desktop key logging data of 6 users is shown in Figure 1. It can be observed that the features' set on desktops is well segregated and poses a relatively simple classification problem. However, once we extracted the same three features from the mobile phone data of 25 users, their feature vectors are extremely diffused as shown in Figure 1(a). Keystroke-based user identification problem is more challenging on mobile phones because they generally have multiplexed keys in a 4×3 matrix. In order to make the data less diffused, we split the feature "digraph" into four types of digraphs as follows:



(a) Plot showing high diffusion in mobile features



(b) Plot showing less diffusion in desktop features

Fig. 1. Plot of features' variation for mobile and desktop

Horizontal Digraph (D_h^a). This is the time elapsed between releasing a key and pressing the adjacent key in the same horizontal row of keys, e.g. the time between key 1 and key 2, key 5 and key 6, key 0 and key * etc.;

Vertical Digraph (D_v^a). This is the time elapsed between releasing a key and pressing the adjacent key in the same vertical column of keys, e.g. the time between key 1 and key 4, key 5 and key 8, key # and key 9 etc.;

Non-adjacent Horizontal Digraph (D_h^{na}). This is the time elapsed between releasing a key and pressing the next in the same horizontal row such that the keys are separated by another key, e.g. time between key 1 and key 3, key 4 and key 6, key * and key # etc.; and

Non-adjacent Vertical Digraph (D_v^{na}). This is the time elapsed between releasing a key and pressing the next in the same vertical column such that the keys are separated by another key, e.g. the time between key 1 and key 7, key 0 and key 5, key 3 and key 9 etc.

Once we extracted these features, we calculated the coefficient of variation (c_v) for each feature to determine variation and randomness in the features' data. From Table 1, we can observe that the coefficient of variation of the key hold time feature for 25 different users is very small (order of 10^{-2}) which highlights that users normally press keys for approximately the same length of time. However, this parameter is significantly higher for digraph times. The coefficient of variation of more than 1 shows large randomness in the data. Therefore, in order to correctly classify a user based on this collective feature set, we need a classifier that can identify classification boundaries for this highly varying data which is a result of diffused usage patterns of different users.

4.2 Accuracy Evaluation of Existing Techniques

As a next logical step, we investigate the accuracy of existing classification schemes, developed for desktop computers, on the mobile phones' dataset. To this end, we evaluate five prominent classifiers proposed in [24],[22],[13],[29],[8],[23]. These classifiers are quite diverse. Naive Bayes [24] is a probabilistic classifier; while Back Propagation Neural Network (BPNN) [22] and Radial Basis Function Network (RBFN) [13] belong to the category of neural networks. In comparison, Kstar [8] is a statistical classifier and J48 [23] is a decision tree classifier. In order to remove any implementation related bias, we have performed our experiments in WEKA [29].

Ideally, we need a classifier that classifies a user as legitimate or imposter with 100% accuracy. In our current accuracy evaluation setup, the errors are of two types: (1) *False Acceptance Rate (FAR)* is defined as the probability that an imposter is classified as a legitimate user, and (2) *False Rejection Rate (FRR)* is defined as the probability that a legitimate user is classified as an imposter.

The results of our experiments are tabulated in Table 2. We can see that the existing classifiers provide an FAR of 30-40% which is not acceptable. Similarly, FRR of most of the classifiers is approximately 30% or more and this again

Table 2. A comparative study of techniques on the basis of key hold time, digraph, and error percentage

Users	Naive Bayes		BPNN		RBFN		Kstar		J48	
	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR
u1	51.1	**6.31**	56.2	12.4	33.2	9.31	**13.2**	22.1	44.3	39.2
u2	32.4	17.9	31.5	58.4	28.4	**11.9**	**11.2**	38.4	31.2	67.4
u3	42.1	11.6	45.3	19.6	**22.5**	**11.2**	34.2	11.5	44.3	26.4
u4	56.9	**7.28**	31.4	11.3	58.3	12.4	49.8	19.5	**21.3**	32.4
u5	33.1	36.6	44.2	24.5	45.2	**21.4**	**32.1**	31.3	33.4	25.8
u6	44.6	17.8	53.4	20.5	48.9	**11.5**	37.6	18.4	**24.6**	32.4
u7	40.2	**21.3**	45.6	18.9	36.7	21.4	43.1	33.4	**21.2**	24.8
u8	29.8	58.2	37.5	23.6	68.9	**18.9**	44.6	26.5	**24.6**	32.1
u9	27.3	62.7	40.4	44.2	44.1	31.3	**24.6**	**21.3**	44.3	38.5
u10	**24.5**	63.2	36.7	72.4	30.9	**43.2**	27.6	53.2	42.1	79.6
u11	41.6	18.9	42.1	19.6	23.5	**12.3**	23.2	31.2	**21.7**	34.5
u12	33.1	37.3	42.1	**28.5**	33.2	33.5	**31.2**	43.2	43.8	73.5
u13	32.1	53.4	42.6	61.3	**19.5**	54.3	24.6	**34.2**	**19.5**	75.2
u14	**22.5**	63.5	28.9	23.1	33.5	31.3	26.6	**21.5**	43.5	39.3
u15	21.5	38.8	33.4	78.9	20.4	59.6	21.3	**31.2**	**12.4**	81.2
u16	43.1	35.8	56.7	19.6	67.5	**15.6**	52.3	33.4	**21.7**	30.4
u17	49.6	**11.9**	61.3	12.4	39.4	13.7	**34.6**	28.5	41.2	23.2
u18	29.8	63.4	31.2	73.2	34.5	35.6	28.7	39.2	**23.5**	**34.6**
u19	52.4	**4.16**	64.7	13.2	**37.4**	15.8	38.6	30.5	47.7	32.4
u20	29.8	**13.2**	**22.5**	38.6	33.3	66.7	27.9	35.4	33.2	28.3
u21	39.8	19.7	53.7	**19.2**	**28.5**	19.8	32.3	31.2	31.4	25.3
u22	39.1	35.6	39.6	44.2	22.1	33.1	**19.4**	31.3	**19.4**	**28.5**
u23	30.9	**23.3**	28.6	26.7	21.8	32.1	**12.5**	43.2	23.4	55.3
u24	33.5	**21.3**	41.4	21.4	31.2	24.1	18.4	22.3	**16.4**	39.2
u25	42.5	19.7	29.6	19.6	38.2	22.4	21.3	38.2	**19.4**	**18.3**
average	36.9	30.5	41.6	32.2	36.0	**26.5**	**29.2**	30.8	29.9	40.7
standard deviation	**9.50**	19.9	11.1	20.8	13.5	15.7	11.0	**9.22**	10.9	19.2

confirms that their accuracies are not acceptable for real-world deployments because such a high FRR will simply frustrate legitimate users.

4.3 Discussion

The main reason for poor FAR and FRR performance of these classifiers is that they are unable to cope with the variation in the feature set that was highlighted by c_v in the previous section. Thus an important outcome of this pilot study is that we need to design a classifier for our user identification and authentication system that should meet the following requirements: (1) it should provide low ($< 5\%$) FAR, (2) it should also provide low ($< 5\%$) FRR, (3) it must have small detection time to correctly classify a user, (4) the system must be deployable on real mobile phones, (5) it should continuously adapt to the variation in the feature set, and (6) it should have low run-time complexity.

Requirement (1) ensures that an imposter does not go undetected. Once this requirement is combined with requirement (3), we reduce the identification delay on a mobile phone. Requirement (2) is important because if our system starts rejecting the legitimate users, it will lead to their frustration and annoyance and the system will lose its usability appeal. Finally requirement (6) is important because a highly complex system can not be deployed on resource constrained mobile phones.

The following section develops a classifier that can simultaneously meet all of the above requirements.

5 A Tri-mode System for Mobile Phone User Identification

Based on the results of the last section, we propose a tri-mode system for mobile phone user identification. To simultaneously cater for the requirements set above, the system operates in three sequential modes.

Learning Mode. This mode can be further divided into two submodes: initial (static) learning and continuous (dynamic) learning. In the static learning phase, a keystroke profile of a user is collected and a feed-forward classifier is trained on this profile. The dynamic learning phase executes continuously to track and learn changes in the user's behavior. These changes are fed back into the feed-forward detector to allow it to adapt to variations in the user's behavior.

Detection Mode. In the detection mode, the classifier trained during the learning mode is used to differentiate between legitimate users and imposters. If the detector raises an alarm during this mode, the system moves to the verification mode.

Verification Mode. In the verification mode, a user is asked to type a remembered 8-character PIN. In the verification mode, we not only compare the typed characters with the stored PIN but also match how the PIN has been typed. In the worst case, when an imposter already knows the PIN, the imposter would still have to enter the PIN using the legitimate user's keystroke dynamics. This mode acts as a final line of defence against an imposter who has successfully breached every other protection layer.

Interested readers can find all the technical details and algorithms used in the development of this tri-mode system in [25]. In subsequent sections, we give a general overview of the algorithms used in each of the modes described above.

5.1 Algorithms in Learning and Detection Modes

Previous results showed that, due to the variation in the feature-set of different users, standard machine learning classifiers cannot provide acceptable error rates for the present problem of keystroke-based mobile phone user identification. Specifically, variation in the features' set results in a diffused dataset and consequently it is not possible to assign crisp classification boundaries to different users. A study of existing classifiers reveals that classifiers based upon *fuzzy logic* [30] are well-suited for such problem. Fuzzy classifiers can provide acceptable accuracies on diffused datasets because they assign a given data point a degree of membership to all available classes. The primary task of fuzzy classification is to determine the boundaries of the decision regions based on the training datapoints. Once the class-labeled decision regions in the feature space are determined, classification of an unknown point is achieved by simply identifying the region in which the unknown point resides. Since fuzzy logic assigns each data point a degree of membership to different decision regions instead of a single association to one decision region (thus showing inherent capability to

deal with fuzzy/diffused datasets), we expect a fuzzy classifier to provide an accurate and efficient learning mechanism for the diffused mobile phone feature-set. The remainder of this section develops and evaluates a fuzzy classifier for mobile phone user classification.

Initial Learning using a Feed-Forward Fuzzy Classifier. We are working on a two-class classification problem as we need to distinguish between a legitimate user and an imposter. A fuzzy system is based on a database, rule base, and a fuzzy inference system. The database is composed of linguistic variables, fuzzy partitions, and membership functions. We now describe our fuzzy clustering algorithm and then evaluate its accuracy on the mobile keystrokes dataset.

In order to determine an initial rule base for fuzzy system, we define the centroid of a cluster in the form of $(x_1, x_2, \ldots, x_z)$, where x_1, x_2, ..., x_z are the values of the first, second, ..., z^{th} feature, respectively, where z is the dimension of the feature vector. It is mentioned earlier that we use $z = 6$ features. For a given data point, we search its value in the corresponding fuzzy sets, determine its degree of membership to each fuzzy partition and then assign the point to the partition with the maximum degree of membership. To determine the consequent of a rule, we find the density of the cluster of the centroid for which we are defining an antecedent of the rule. If a cluster has *high*, *medium* or *low* density then the output belongs to the fuzzy partitions *high*, *medium* or *low*, respectively, in the consequent of the rule. We repeat this procedure for all training data points to define a rule-base using the centroids of all the clusters.

To give a preliminary indication of the accuracy of the first phase of our proposed system, the FAR and FRR values of the fuzzy classifier are shown in Table 3. FAR and FRR of approximately 18.6% and 19.0%, respectively – much better

Table 3. A comparative study of the feasible techniques

Users	RBFN		Fuzzy		PSO-Fuzzy		GA-Fuzzy		PSO-GA Fuzzy	
	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR
u1	33.2	9.31	18.1	19.3	8.34	7.55	8.32	8.54	**2.13**	**1.76**
u2	28.4	11.9	17.3	21.6	9.63	6.43	8.73	8.11	**1.61**	**0.82**
u3	22.5	11.2	21.3	17.5	7.43	9.22	6.34	7.63	**2.14**	**1.71**
u4	58.3	12.4	17.6	16.2	7.92	8.74	8.91	6.19	**1.19**	**1.56**
u5	45.2	21.4	18.3	15.3	8.73	6.12	7.43	8.11	**1.87**	**2.01**
u6	48.9	11.5	17.1	18.2	9.54	9.01	8.63	7.23	**2.01**	**2.33**
u7	36.7	21.4	18.9	19.9	6.94	5.91	7.23	3.71	**1.46**	**2.15**
u8	68.9	18.9	21.6	21.4	7.22	6.42	8.34	9.73	**2.14**	**1.61**
u9	44.1	31.3	19.3	19.8	9.02	6.71	9.84	7.29	**3.34**	**1.14**
u10	30.9	43.2	17.3	21.2	11.4	9.13	10.1	8.92	**1.73**	**1.28**
u11	23.5	12.3	18.3	22.1	7.44	8.21	9.23	9.31	**2.43**	**1.86**
u12	33.2	33.5	16.2	17.2	5.22	9.42	9.31	8.22	**1.71**	**1.92**
u13	19.5	54.3	19.7	18.1	8.23	6.12	8.34	9.31	**3.44**	**1.81**
u14	33.5	31.3	17.3	16.4	9.15	9.84	8.91	8.34	**1.29**	**1.38**
u15	20.4	59.6	17.3	16.3	7.97	8.92	7.25	8.81	**3.37**	**1.95**
u16	67.5	15.6	18.1	16.9	6.95	7.01	6.33	9.42	**2.31**	**2.11**
u17	39.4	13.7	19.2	14.5	9.12	9.21	8.93	7.71	**1.82**	**2.04**
u18	34.5	35.6	19.1	22.4	10.1	7.01	11.3	8.73	**1.01**	**1.72**
u19	37.4	15.8	15.1	26.8	6.02	5.17	9.61	7.29	**1.21**	**1.04**
u20	33.3	66.7	17.5	18.1	9.14	5.95	7.21	6.32	**2.04**	**1.33**
u21	28.5	19.8	22.1	17.2	7.05	5.11	8.87	9.82	**1.41**	**2.38**
u22	22.1	33.1	19.2	15.5	6.21	9.31	9.94	9.63	**2.12**	**2.24**
u23	21.8	32.1	19.8	22.6	9.11	8.01	12.1	8.24	**2.02**	**2.92**
u24	31.2	24.1	16.6	21.8	6.22	6.16	10.4	7.31	**3.11**	**1.14**
u25	38.2	22.4	22.1	19.3	8.34	8.91	8.22	4.72	**2.97**	**1.19**
Avg	36.0	26.5	18.6	19.0	8.09	7.58	8.79	7.94	**2.07**	**1.73**
standard deviation	13.5	15.7	1.86	3.00	1.47	1.55	1.46	1.56	**0.73**	**0.47**

compared with existing classifiers in Table 2 – are still far from acceptable. These accuracy results do not meet the requirements that we have set for our system. In our performance evaluation, we observed that the main accuracy limiting factor for the fuzzy classifier was the dynamically changing keystroke behavior of mobile phone users. Thus the performance of the feed-forward fuzzy classifier can be improved if we use an online dynamic optimizer that can dynamically track and provide the changing feature trends as feedback into the fuzzy system.

Prior work has shown that Particle Swarm Optimization (PSO) and Genetic Algorithms (GAs) have the capability to adapt with changes in datasets [10],[4]. Therefore, in subsequent sections, we study the effect of incorporating a PSO- and GA-based optimizer into the fuzzy classifier.

Continuous Learning using Dynamic Optimizers. As mentioned above, after an initial rule base is generated, we use Particle Swarm Optimization (PSO) and Genetic Algorithms (GAs) to adapt the rule base to dynamically varying user keystoke behavior.

Particle Swarm Optimization (PSO). The main idea of PSO [16] is to use a swarm of agents that is spread on the landscape of search space, and these agents, through local interactions, try to find an optimal solution to the problem. The characteristic that makes PSO successful is the communication between the agents which allows agents to converge to the best location. Table 3 tabulates the results of our fuzzy classifier optimized using PSO. It can be seen that the FAR and FRR values have improved significantly to approximately 8% (averaged). However, even after this improvement, a system with an error rate of around 8% is not usable in the real-world.

Genetic Algorithm (GA). Genetic algorithms are well-known for providing acceptable solutions to dynamic optimization problems [4]. Unlike PSO, GA does not utilize feedback explicitly; rather, it uses genetic operators of selection, crossover and mutation to find the best solution. Use of GA reduces error rate to approximately 8% (averaged) which is almost the same as obtained by the fuzzy classifier optimized using PSO.

Hybrid PSO-GA. PSO utilizes the concept of feedback and GA uses the diversity achieved by randomness. Both of these optimizers have improved FAR and FRR considerably. If we can somehow combine the concept of feedback with randomness, theoretically the accuracy of our fuzzy classifier should improve. For this scenario, we use PSO and GA together for optimizing the database and rule base of the feed-forward fuzzy classifier. The results of the fuzzy classifier optimized by a hybrid PSO-GA optimizer are tabulated in Table 3. It can be seen that the FAR and FRR have improved substantially to approximately 2%; as a result, our hybrid system is able to meet the accuracy requirements set earlier.

Another important thing to mention here is the standard deviation of the results. The standard deviation of our proposed hybrid PSO-GA-Fuzzy system is only 0.73% for FAR and 0.47% for FRR which is negligible. We have repeated the experiments for our scheme 500 times and the confidence interval of our

results is 95% using the t-distribution. This shows that the results produced by our system are statistically significant and the variation in the results is significantly small.

5.2 Algorithm in Verification Mode

If the detection mode raises an alarm, the system moves to the verification mode. In this mode, we ask the suspicious user to enter a remembered 8-character PIN. During the PIN entry process, we observe his/her keystroke patterns and conclude whether or not the current user is an imposter. In this mode, the system extracts three features – key hold time, digraph (irrespective of the position of keys), and error rate – from the key log of the PIN entry process. Note that here we use only three features because we have empirically determined that these features are sufficient to achieve approximately 0% error rate.

We have also empirically concluded a rule: if a potential imposter passes the test twice in three attempts, we declare him/her a legitimate user. We have arrived at this configuration by running a controlled experiment. We asked 10 of our colleagues to enter their PINs 30 times for training. After training, we asked all of these 10 colleagues to enter their passwords 5 times. We observed that each of them was able to enter his/her password with correct behavior at least two out of the first three attempts. Later, we selected three imposters for each of those 10 colleagues and told them the correct passwords of respective legitimate users. We again requested imposters to enter the password 5 times and it was interesting to note that none of them was able to successfully enter the password with a correct keystrokes pattern even once.

For PIN verification, we have designed a simple, efficient and accurate classifier specifically for keystroke dynamics. The motivation behind developing a new classifier for PIN verification mode was that in case of PIN verification we already know the correct PIN and consequently we know what to expect from the user. Thus a classifier with significantly small computational complexity can perform this task. Our classifier dynamically assigns an impression coefficient (iC) to a user on the basis of his/her PIN typing pattern. We argue that a legitimate user is less likely to commit a mistake while entering his/her PIN; therefore, committing a mistake during the PIN entry process counts negatively towards the possibility that the current user is the legitimate user. We calculate the difference between the key hold times of keys of current profile with the key hold times of all the corresponding keys of the standard profiles of a user and then sum up all these differences to find the overall difference in the key hold time. Similarly, we find an overall difference in the digraph time. Finally, we sum overall key hold time difference and digraph difference to define the impression coefficient of PIN entry process. If a user commits a mistake during the PIN entry process, we penalize him/her for each error by adding l milliseconds to the overall difference value.

The overall difference is compared with a threshold value that is also dynamically calculated. If iC of a user is larger than this threshold value, we classify him as an imposter otherwise he/she is a legitimate user. It is important to emphasize that

we do not train our system with imposters' profiles. The mathematical explanation of our proposed classifier is given in the following text.

The size of the PIN is s characters, the number of keys pressed by the user to enter s characters is represented by t, and the number of training profiles is represented by n. P^k is a matrix consisting of n rows corresponding to n training profiles and t columns corresponding to t key hold times. P^{uk} is a row vector of t columns; each column corresponds to a key hold time for a key press in an unknown profile. D^k, similarly, is a matrix of dimensions $n \times t-1$ for digraph times obtained from training profiles and D^{uk} is a row vector of $t-1$ columns representing the digraph times of an unknown user. The mathematical model is given in following equations:

$$iC = \sum_{i=1}^{n} \left(\sum_{j=1}^{t} \left| p_{ij}^{k} - p_{j}^{uk} \right| + \sum_{j=1}^{t-1} \left| d_{ij}^{k} - d_{j}^{uk} \right| \right) + e \times l, \tag{1}$$

$$\mu = \frac{\sum_{m=1}^{n} \left\{ \sum_{i=1}^{n} \left(\sum_{j=1}^{t} \left| p_{ij}^{k} - p_{mj}^{k} \right| + \sum_{j=1}^{t-1} \left| d_{ij}^{k} - d_{mj}^{k} \right| \right) \right\}}{n}, \tag{2}$$

$$\sigma = \left[\sum_{m=1}^{n} \left\{ \sum_{i=1}^{n} \left(\sum_{j=1}^{t} \left| p_{ij}^{k} - p_{mj}^{k} \right| + \sum_{j=1}^{t-1} \left| d_{ij}^{k} - d_{mj}^{k} \right| \right) - \mu \right\}^{2} / n \right]^{\frac{1}{2}}, \tag{3}$$

where $p_{ij}^k, p_{mj}^k \in P^k$; $p_j^{uk} \in P^{uk}$; $d_{ij}^k, d_{mj}^k \in D^k$; $d_j^{uk} \in D^{uk}$; and e represents the number of backspaces during PIN entry. Moreover, if $iC \geq \mu + a\sigma$ then the user is classified as an imposter; otherwise the user is classified as a legitimate user. We have empirically determined that values of $l = 5$ and $a = 3$ provide 0% error. The following section evaluates the accuracy of the proposed tri-mode system for varying system parameters.

6 Performance Evaluation

In this section, we first evaluate the accuracy of the proposed system for a fixed training profile. We then systematically evaluate the system's performance for different parameters. Specifically, we chronologically answer the following questions: (1) What is the accuracy of the system for a fixed profile?, (2) What is the impact of number of profiles on the accuracy of our system?, (3) What is the relationship between the size of a profile and the accuracy of our system?, (4) What is the average user identification delay in terms of mobile phone usage (we report it in terms of the number of SMSs)?, (5) How much damage an imposter can do in 250 keystrokes?, and (6) What are the training and testing times of our system?

What is the accuracy of the system for a fixed profile? The accuracy of our system can be viewed from the Table 4. It can be seen that our tri-mode

Table 4. Accuracy results after detection mode and verification mode for a fixed profile size of 250 keystrokes

Users	After Detection Mode		After Verification Mode		Users	After Detection Mode		After Verification Mode	
	FAR	FRR	FAR	FRR		FAR	FRR	FAR	FRR
u1	2.13	1.76	2.13	0	**u2**	1.61	0.82	1.61	0
u3	2.14	1.71	2.14	0	**u4**	1.19	1.56	1.19	0
u5	1.87	2.01	1.87	0	**u6**	2.01	2.33	2.01	0
u7	1.46	2.15	1.46	0	**u8**	2.14	1.61	2.14	0
u9	3.34	1.14	3.34	0	**u10**	1.73	1.28	1.73	0
u11	2.43	1.86	2.43	0	**u12**	1.71	1.92	1.71	0
u13	3.44	1.81	3.44	0	**u14**	1.29	1.38	1.29	0
u15	3.37	1.95	3.37	0	**u16**	2.31	2.11	2.31	0
u17	1.82	2.04	1.82	0	**u18**	1.01	1.72	1.01	0
u19	1.21	1.04	1.21	0	**u20**	2.04	1.33	2.04	0
u21	1.41	2.38	1.41	0	**u22**	2.12	2.24	2.12	0
u23	2.02	2.92	2.02	0	**u24**	3.11	1.14	3.11	0
u25	2.97	1.19	2.97	0	**Avg**	2.07	1.73	2.07	0
SD	0.73	0.47	0.73	0	—	—	—	—	—

system achieves 0% FRR and approximately 2% FAR after the verification mode. 0% FRR indicates that our system is completely user friendly and never rejects a legitimate user. It also has a very low FAR as compared to other techniques.

What is the impact of number of profiles on the accuracy of our system? Scalability analysis is important to determine the minimum number of profiles/keystrokes required to achieve acceptable accuracy. We take the users with the most number of profiles (u10, u14, and u15) for our scalability analysis and tabulate the results in Table 5. Note that each profile is made up of 250 keys. The results in Table 5 suggest a gradual, almost linear decrease in FAR and FRR as we increase the number of training profiles up to 50. This shows that as the number of training profiles increases, the accuracy of our system increases.

What is the relationship between the size of a profile and accuracy of our system? For the same users (u10, u14, and u15,) we now take 50 profiles of each user and study the relationship between FAR and FRR and the size of a profile. These results are also tabulated in Table 5. It is obvious from Table 5 that FAR and FRR values degrade for small size profiles, however, for a profile of 250 keys, the error rates on average are 2%. It can also be seen that increasing the size of profile from 250 to 350 keys further improves the detection accuracy but the improvement is not much significant; therefore, we use a profile of 250 keys. Note that increasing the size of profile not only increases the detection accuracy but also the time required to make a profile. Our aim is to get reasonable detection accuracy with as small a profile size as possible. Profile size of 250 keys satisfies the criteria.

Table 5. Relationship of number of training profiles and size of a profile with error rates

Users	**Number of profiles** (Profile Size = 250)								**Size of profile** (Number of Profiles = 50)									
	20		**30**		**40**		**50**		**150**		**200**		**250**		**300**		**350**	
	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR
u10	2.32	1.99	2.01	1.51	1.93	1.35	**1.74**	**1.29**	11.2	7.28	4.98	3.45	1.74	1.29	1.45	1.11	**1.10**	**1.01**
u14	3.21	2.21	1.97	2.01	1.77	1.78	**1.30**	**1.40**	9.21	8.12	4.11	4.01	1.30	1.40	1.03	1.21	**0.97**	**1.11**
u15	5.89	3.13	5.11	2.72	4.01	2.11	**3.39**	**1.98**	17.8	11.5	9.62	6.22	3.39	1.98	2.87	1.23	**1.91**	**0.99**

Table 6. User identification delay

Users	Avg SMSs / Profile	Users	Avg SMSs / Profile	Users	Avg SMSs / Profile	Users	Avg SMSs / Profile	Users	Avg SMSs / Profile	Users	Avg SMSs / Profile	Users	Avg SMSs / Profile
u1	1.06	**u2**	1.44	**u3**	1.25	**u4**	1.09	**u5**	1.26	**u6**	0.65	**u7**	1.03
u8	1.03	**u9**	1.11	**u10**	1.34	**u11**	1.04	**u12**	1.27	**u13**	1.32	**u14**	1.39
u15	1.33	**u16**	0.16	**u17**	1.17	**u18**	1.15	**u19**	1.00	**u20**	1.43	**u21**	1.25
u22	1.22	**u23**	1.24	**u24**	1.13	**u25**	1.30	—	—	—	—	—	—

What is the user identification delay? Table 6 shows the average number of SMS a user types in a single profile. Remember our system tries to classify a user after every 250 keystrokes using the keystrokes features' set. It can be seen from Table 6 that on the average a profile of 250 keystrokes is generated once a user sends just 1 SMS. So our detection delay is bounded by the time within which a user sends an SMS. Detection delay is not a significant problem if the objective of an imposter is to steal the mobile phone. However, this delay will become very crucial if an imposter wants to steal information from the mobile phone. We invoke the verification mode of our system to disallow an imposter to transmit data from the mobile phone.

How much damage can an imposter do in 250 keystrokes? We have done an interesting study in which we requested 4 of our colleagues in the virology lab to act as imposters on a trained mobile phone to get an understanding of how much data they can read in a given document. (Remember that they cannot upload or copy the data because of the verification mode). We downloaded the current paper (20 pages long) on a smart phone and told the imposters the exact directory path of the paper. (Remember that it is the best case scenario for an imposter; otherwise, he needs to press more keys to search a document). We asked them to find the line "and the system will lose its usability appeal" in the paper that happens to be on page 9. We tabulate the number of keys pressed by each of them in Table 7 to locate the required information. It is interesting to note that, even in this best case scenario, only one of them was able to locate the information within 250 keystrokes. We have also done an interesting study to understand that how far different imposters can scan the given document in 250 keystrokes. **i4** appears to be a smart imposter who manages to reach page 14 in 250 keystrokes. Most of the users pressed 250 keystrokes in 8 to 15 minutes.

What are the training and testing times of our system? We now analyze the training and testing times of different classifiers in Table 8. The training time of our classifier is 28 seconds, but our testing time is just 520 milliseconds. Thus, while the system's run-time complexity is comparable to other existing algorithms, its training complexity is significantly higher. The main source of this

Table 7. Analysis showing the number of keystrokes to perform the task

Imposters	Number of Keys to Perform the Task	Page # After 250 Keys	Imposters	Number of Keys to Perform the Task	Page # After 250 Keys
i1	332	7	**i2**	442	6
i3	297	8	**i4**	189	14

Table 8. The processing overheads of classifiers on an old 233MHz, 32MB RAM computer

Algorithm	Train (*secs*)	Test (*secs*)	Algorithm	Train (*secs*)	Test (*secs*)	Algorithm	Train (*secs*)	Test (*secs*)
PSO-GA Fuzzy	28	0.52	**Naive Bayes**	0	0.52	**BPNN**	4.8	2.0
RBFN	0.41	0.42	**Kstar**	8	0.21	**J48**	0.23	0.22

complexity are the back-end dynamic optimizers used in our system. However, we emphasize that training complexity needs to be incurred very infrequently after every 5 training profiles (usually after a few hours). Moreover, unlike desktop computers, mobile phones remain idle much of the time and the retraining can be performed during these inactivity periods.

7 Limitations and Potential Countermeasures

We now highlight the important limitations and countermeasures of our system.

Identification delay period. Our system can detect an imposter after observing a minimum of 250 keystrokes. The identification delay is hence a function of the imposters keyboard usage. We argue that an imposter's keyboard usage can belong to one of the following two types: (1) he/she wants to get access to the sensitive information/documents on the phone, and (2) he/she wants to steal the mobile phone. In the first case, the imposter must try to quickly get access to the sensitive information and, as a result, the time to generate a profile of 250 keystrokes, as mentioned before, will reduce to 10-15 minutes. If the imposter is of the second type, then our system will detect him/her after 250 keystrokes through our PIN verification procedure.

Accuracy is sensitive to the number of profiles. Another shortcoming of our approach is that it requires a cold start of 30 or more profiles to accurately learn the behavior of a user. In this time period, the system might suffer from relatively high FAR and FRR which are still comparable with the existing techniques (see Tables 2 and 5). But our system provides significantly better FAR and FRR after collecting just one week of training data, which we believe is quite reasonable.

Portability to full keyboard smart phones. We have not tested our prototype on BlackBerry category of phones that have QWERTY keyboards. While we believe that the results of our system will scale to these phones, we are currently soliciting volunteers with full keyboard Nokia phones for testing and evaluation.

Relatively large training time. Our systems takes 28 seconds on the average once we retrain it after every 5 profiles. During these 28 seconds after every few hours, the response time of the mobile phone degrades which might result in some annoyance to the user. We argue that this cost is worth the benefit of very low FAR and FRR values of our system. Moreover, as suggested earlier, the retraining module can be customized to execute during inactivity periods.

Table 9. Improvement shown by Hybrid PSO-GA Fuzzy system over the other classifiers

Algorithm	%Improvement		Algorithm	%Improvement	
	FAR	FRR		FAR	FRR
Naive Bayes	94.4	94.3	**BPNN**	95.0	94.6
RBFN	94.2	93.5	**Kstar**	92.9	94.4
J48	93.1	95.7	**Fuzzy**	88.8	89.1
PSO-Fuzzy	74.4	77.2	**GA-Fuzzy**	76.4	78.2

Resilience to reinstalling OS. A savvy imposter may reinstall the OS on the phone, thus circumventing our system. This is a common limitation for all host-based intrusion detection systems. A solution to this problem is OS virtualization which is computationally infeasible on contemporary mobile phones.

8 Conclusion and Future Work

We have proposed a user identification system that monitors the keystroke dynamics of a mobile phone user to differentiate legitimate users from imposters. We have used a custom dataset of 25 diverse mobile phone users to show that the proposed system can provide an error rate of less than 2% after the detection mode and an FRR of close to zero after the PIN verification mode. We have also compared our approach with 5 state-of-the-art existing techniques for keystroke-based user identification. Table 9 shows percentage improvement of our scheme compared to the existing schemes on our dataset. In future, we intend to incorporate our system in a Symbian mobile phone and then evaluate its accuracy on-line under real-world usage circumstances. Interested researchers can also work on the modification of the proposed system for its portability to the mobile phones with QWERTY keyboards.

Acknowledgments

Saira Zahid and Muhammad Shahzad are working on the project "An Intelligent Secure Kernel for Next Generation Mobile Computing Devices"[7] and are in part funded by the National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan. The information, data, comments, and views detailed herein may not necessarily reflect the endorsements of views of the National ICT R&D Fund.

References

1. Red herring mobiles scream for help: Uk-based mobile security company adds security to mobile phones (October 2006)
2. Babin, S., Pranata, A.: Developing Software for Symbian OS: A Beginner's Guide to Creating Symbian OS V9 Smartphone Applications in C++. John Wiley & Sons, Chichester (2007)

[7] `http://isk.nexginrc.org/`

3. Bleha, S., Slivinsky, C., Hussien, B.: Computer-access security systems using keystroke dynamics. IEEE Transactions on Pattern Analysis and Machine Intelligence 12(12), 1217–1222 (1990)
4. Branke, J.: Evolutionary Optimization in Dynamic Environments. Kluwer Academic Publishers, Dordrecht (2002)
5. Card, S.K., Moran, T.P., Newell, A.: Computer text-editing: An information-processing analysis of a routine cognitive skill (1987)
6. Clarke, N.L., Furnell, S.M.: Authentication of users on mobile telephones–A survey of attitudes and practices. Computers & Security 24(7), 519–527 (2005)
7. Clarke, N.L., Furnell, S.M.: Authenticating mobile phone users using keystroke analysis. International Journal of Information Security 6(1), 1–14 (2007)
8. Cleary, J.G., Trigg, L.E.: K*: An Instance-based Learner Using an Entropic Distance Measure. In: Machine Learning-International Workshop then Conference, pp. 108–114. Morgan Kaufman Publishers, Inc., San Francisco (1995)
9. Corner, M.D., Noble, B.D.: Zero-interaction authentication. In: Proceedings of the 8th annual international conference on Mobile computing and networking, pp. 1–11. ACM, New York (2002)
10. Engelbrecht, A.P.: Fundamentals of computational swarm intelligence. John Wiley & Sons, Chichester (2006)
11. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
12. BioPassword Inc., http://www.biopassword.com
13. Hwang, Y.S., Bang, S.Y.: An Efficient Method to Construct a Radial Basis Function Neural Network Classifier. Neural Networks 10(8), 1495–1503 (1997)
14. Joyce, R., Gupta, G.: Identity authentication based on keystroke latencies. Communications of the ACM 33(2), 168–176 (1990)
15. Karatzouni, S., Clarke, N.: Keystroke Analysis for Thumb-based Keyboards on Mobile Devices. In: International Federation for Information Processing-Publications-IFIP, vol. 232, pp. 253–263 (2007)
16. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of IEEE International Conference on Neural Networks, 1995, vol. 4 (1995)
17. Leggett, J., Williams, G.: Verifying identity via keystroke characteristics. International Journal of Man-Machine Studies 28(1), 67–76 (1988)
18. Leggett, J., Williams, G., Usnick, M., Longnecker, M.: Dynamic identity verification via keystroke characteristics. International Journal of Man-Machine Studies 35(6), 859–870 (1991)
19. Lilley, P.: Hacked, Attacked & Abused: Digital Crime Exposed. Kogan Page Ltd. (2002)
20. Mahar, D., Napier, R., Wagner, M., Laverty, W., Henderson, R.D., Hiron, M.: Optimizing digraph-latency based biometric typist verification systems: inter and intra typist differences in digraph latency distributions. International Journal of Human-Computer Studies 43(4), 579–592 (1995)
21. Obaidat, M.S., Sadoun, B.: Keystroke Dynamics based Authentication Biometrics. Systems, Man and Cybernatics 27(2), 261–269 (1997)
22. Paola, J.D., Schowengerdt, R.: A detailed comparison of backpropagation neural network and maximum-likelihood classifiers for urban land use classification. IEEE Transactions on Geoscience and Remote Sensing 33(4), 981–996 (1995)
23. Quinlan, J.R.: Bagging, Boosting, and C4. 5. In: Proceedings of the NCAI, pp. 725–730 (1996)
24. Rish, I.: An empirical study of the naive Bayes classifier. In: Proceedings of IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence, vol. 335 (2001)

25. Shahzad, M., Zahid, S., Farooq, M.: A Hybrid GA-PSO Fuzzy System for User Identification on Smart Phones. In: Proceedings of the 11th annual conference on Genetic and evolutionary computation, Montreal, Canada. ACM, New York (in press, 2009)
26. Sims, D.: Biometric recognition: our hands, eyes, and faces give us away. IEEE Computer Graphics and Applications 14(5), 14–15 (1994)
27. Umphress, D., Williams, G.: Identity Verification Through Keyboard Characteristics. International Journal of Man-Machine Studies 23(3), 263–273 (1985)
28. Wang, X., Heydari, M.H., Lin, H.: An intrusion-tolerant password authentication system. In: Computer Security Applications Conference, 2003. Proceedings. 19th Annual, pp. 110–118 (2003)
29. Witten, I.H.: University of Waikato, and Dept. of Computer Science. WEKA Practical Machine Learning Tools and Techniques with Java Implementations. Dept. of Computer Science, University of Waikato (1999)
30. Zadeh, L.A.: Fuzzy sets. Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers (1996)

VirusMeter: Preventing Your Cellphone from Spies

Lei Liu[1], Guanhua Yan[2], Xinwen Zhang[3], and Songqing Chen[1]

[1] Department of Computer Science
George Mason University
[2] Information Sciences Group (CCS-3)
Los Alamos National Laboratory
[3] Computer Science Lab
Samsung Information Systems America

Abstract. Due to the rapid advancement of mobile communication technology, mobile devices nowadays can support a variety of data services that are not traditionally available. With the growing popularity of mobile devices in the last few years, attacks targeting them are also surging. Existing mobile malware detection techniques, which are often borrowed from solutions to Internet malware detection, do not perform as effectively due to the limited computing resources on mobile devices.

In this paper, we propose VirusMeter, a novel and general malware detection method, to detect anomalous behaviors on mobile devices. The rationale underlying VirusMeter is the fact that mobile devices are usually battery powered and any malicious activity would inevitably consume some battery power. By monitoring power consumption on a mobile device, VirusMeter catches misbehaviors that lead to abnormal power consumption. For this purpose, VirusMeter relies on a concise user-centric power model that characterizes power consumption of common user behaviors. In a real-time mode, VirusMeter can perform fast malware detection with trivial runtime overhead. When the battery is charging (referred to as a battery-charging mode), VirusMeter applies more sophisticated machine learning techniques to further improve the detection accuracy. To demonstrate its feasibility and effectiveness, we have implemented a VirusMeter prototype on Nokia 5500 Sport and used it to evaluate some real cellphone malware, including FlexiSPY and Cabir. Our experimental results show that VirusMeter can effectively detect these malware activities with less than 1.5% additional power consumption in real time.

Keywords: mobile malware, mobile device security, anomaly detection, power consumption.

1 Introduction

With the ever-improving chip design technology, the computing power of microprocessors is continuously increasing, which enables more and more features on mobile devices that were not available in the past. For example, today many

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 244–264, 2009.
© Springer-Verlag Berlin Heidelberg 2009

cellphones come with various data services, such as text messaging, emailing, Web surfing, in addition to the traditional voice services. Due to their all-in-one convenience, these increasingly powerful mobile devices are gaining a lot of popularity: It has been expected a mobile population of 5 billion by 2015 [1] and out of 1 billion camera phones to be shipped in 2008, smartphones represent about 10% of the market or about 100 million units [2]. Moreover, the new generation of mobile devices provide a more open environment than their ancestors. They now can not only run sandbox applications shipped from original manufacturers, but also install and execute third-party applications that conform to the norms of their underlying operating systems.

The new features brought by exotic applications, although rendering mobile devices more attractive to their users, also open the door for malicious attacks. By the end of 2007, there were over 370 different mobile malware in the wild [21]. The debut of Cabir [3] in 2004, which spreads through Bluetooth connections, is commonly accepted as the inception of modern cellphone virus [4]. Since then, a number of malware instances have been found exploiting vulnerabilities of mobile devices, such as Cabir [9] and Commwarrior [8]. These mobile malware have created serious security concerns to not only the mobile users, but also the network operators, such as information stealing, overcharging, battery exhaustion, and network congestion.

Despite the immense security threats posed by mobile malware, their detection and defense is still lagging behind. Many signature- and anomaly-based schemes for IP networks have been extended for mobile network malware detection and prevention [12,30,31]. For example, Hu and Venugopal proposed to extract signatures from mobile malware samples and then scan network traffic for these signatures [20]. Similar to their counterparts on IP networks, however, signature-based approaches can easily be circumvented by various techniques, such as encryption, obfuscation, and packing. On the other hand, anomaly-based detection schemes often demand accurate and complete models for normal states and are thus prone to high false alarm rates.

Recently, behavioral signatures have also been proposed for mobile malware detection [10]. They have their own limitations. On one hand, monitoring API calls within an emulated environment and running sophisticated machine learning algorithms for detection are hardly practical for resource-constrained mobile devices due to high detection overhead, not mentioning that most manufacturers do not publicize all relevant APIs on commodity mobile devices. On the other hand, stealthy malware can mimic user behavior or hide its activities among normal user activities to evade detection by API tracking. For example, FlexiSPY[5]-like malware that perform eavesdropping does not show anomalies in the order of relevant API calls, since they are typically implemented as if the user has received an incoming call. To detect energy-greedy malware and variants of existing malware, Kim et al. proposed to use power signatures based on system hardware states tainted by known malware [22]. Their approach, however, is mainly useful for detecting known malware and their variants.

In this study, we propose VirusMeter, a novel and general mobile malware detection method, to detect malware on mobile devices without demanding external support. The design of VirusMeter is based on the fact that *mobile devices are commonly battery powered and any malware activity on a mobile device will inevitably consume battery power*. VirusMeter monitors and audits power consumption on mobile devices with a behavior-power model that accurately characterizes power consumption of normal user behaviors. Towards this goal, VirusMeter needs to overcome several challenges. *First*, VirusMeter requires a power model that can accurately characterize power consumption of user behaviors on mobile devices, but such a model is not readily available as yet. *Second*, VirusMeter needs to measure battery power in real time. Existing research however shows that precise battery power measurement is difficult due to many electro-chemical properties. In addition, although in practice mobile devices commonly have battery power indicators, their precision varies significantly from device to device. Examining the battery capacity frequently also incurs high computational overhead, rendering it hardly practical in reality. *Third*, as VirusMeter aims to run on on-the-shelf mobile devices without external support, it must be lightweight itself, without consuming too much CPU (and thus battery power); otherwise, it can adversely affect the detection accuracy.

To overcome these challenges, we design a user-centric power model that, as opposed to a system-centric model which requires in-depth understanding of various system-level behaviors and states, has only a small number of states based on common user operations. VirusMeter is designed to run in two modes: It, when in a *real-time* detection mode, performs fast malware detection, but when in a *battery-charging* mode, applies advanced machine learning techniques to detect stealthy malware with high accuracy.

To demonstrate its feasibility and effectiveness, we implement a VirusMeter prototype on Nokia 5500 Sport and evaluate its performance with real-world smartphone viruses including Cabir and FlexiSPY. The results show that VirusMeter can effectively detect the malware by consuming less than 1.5% additional power in a real-time mode. In a battery-charging mode, VirusMeter, by using advanced machine learning techniques, can considerably improve the detection rate up to 98.6%.

The remainder of the paper is organized as follows. Some related work is presented in section 2. We overview the VirusMeter design in section 3. We present our model designs, data collection, and model checking for VirusMeter in sections 4, 5, 6, respectively. We present our implementation in section 7 and evaluation results in section 8. We discuss some limitations and future work in section 9 and make concluding remarks in section 10.

2 Related Work

The increasing popularity of mobile devices with faster microchips and larger memory space has made them a lucrative playground for malware spreading. Existing studies [21] show that there are more than 370 mobile malware in the

wild. As Symbian occupies the largest cellphone OS market share, it has been targeted by most of these mobile malware. Different approaches have been employed to classify existing mobile malware. For instance, mobile viruses have been classified based on their infection vectors, such as Bluetooth, MMS, memory cards, and user downloading [13,21]. Currently, user downloading, Bluetooth, and MMS are the most popular channels for mobile malware propagation. Several studies on mobile malware have focused on understanding their propagation behaviors. For example, an agent-based model has been developed to study worms spreading over short-range radio and cellular messaging systems [11]. A probabilistic queuing model is proposed for the spreading of mobile worms over wireless connections [23], and a detailed mathematical model is also developed in [32] to characterize specifically the propagation process of Bluetooth worms. To detect Bluetooth worm outbreaks, Su et al. proposed to deploy monitors in high-traffic areas [29]. To simulate worm propagation in mobile phone networks, Fleizach et al. [17] developed a simulator with great details, including realistic topologies, provisioned capacities of cellular networks, and realistic contact graphs.

Some early studies on defense schemes against mobile malware have mainly focused on understanding their attack characteristics. For example, various potential attacks from a compromised cellphone and the corresponding defenses have been studied [15,16,19,26]. An algorithm based on user interactions is proposed to identify vulnerable users [12]. In [30], several schemes have been studied to mitigate DoS attacks via queuing in the network. To prevent cross service boundary attacks, a labeling technique is used to separate the phone interface from the PDA interface of a mobile device [24]. Sarat et al. [27] proposed to integrate commonwalk lengths and node frequencies to detect worms and determine their propagation origins. Recently, SmartSiren [13] showed how to use a proxy to detect malware by analyzing collected user communication logs. Bose et al. [10] proposed to extract behavioral signatures for mobile malware detection.

So far, existing schemes either have limited effectiveness by targeting particular situations (such as attacks through SMS), and/or demand significant infrastructure support, and/or demand non-trivial computing resources from mobile devices. By contrast, VirusMeter is a general approach, regardless of how malware invade into a system or whether they are known in advance. VirusMeter is also lightweight and can run on a mobile device without any external support. A previous approach that also aims to detect energy-greedy anomalies [22] is closest to VirusMeter. However, it is only effective to detect known malware and their variants, and works only for a single process mode which stealthy malware can easily evade by activating itself on when a user process is active.

3 Overview of VirusMeter Design

The rationale behind VirusMeter is the fact that any malware activities on a mobile device must consume some battery power. Hence, abnormal battery power consumption is a good indicator that some misbehavior has been conducted. Accordingly, VirusMeter monitors battery power usage on a mobile device and

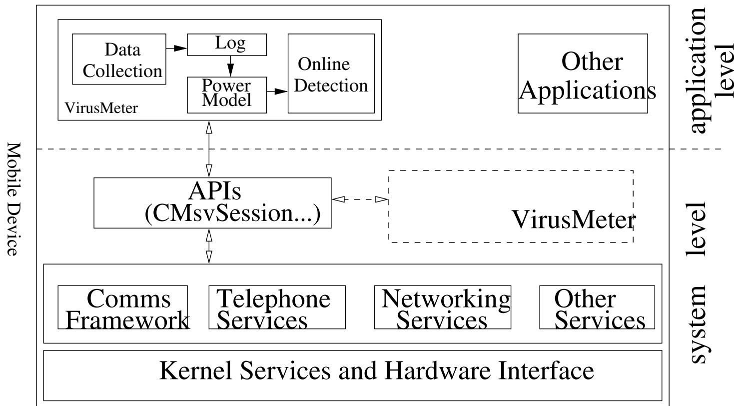


Fig. 1. VirusMeter Runs on a Mobile Device

compares it against a pre-defined power consumption model to identify abnormal activities due to mobile malware.

Figure 1 shows the work flow of VirusMeter when executed on a mobile device. VirusMeter can run at either the system level or the application level (our current implementation is at the application level). Running at the system level is more robust against attacks since mobile OSes, such as Symbian and Windows Mobile, are often only accessible to device manufacturers or authorized parties. As shown in the figure, VirusMeter, using APIs provided by the underlying mobile OS, collects necessary information of supported services as well as the current remaining battery capacity. VirusMeter, based on the pre-defined power model, calculates how much power could have been consumed due to these services and then compares it against the actually measured power consumption. The comparison result portends whether abnormal power draining has occured: If the difference exceeds a pre-specified threshold, VirusMeter raises an alarm indicating the existence of potential malware. Such comparison can be done in real time (a real-time mode) for fast malware detection, or when the battery is charging (a battery-charging mode) for high detection accuracy.

The alarms raised by VirusMeter are instrumental to further revealing malicious activities of the mobile malware. For instance, the user can check the communication records of her mobile device provided by the network operator to see whether there are any suspicious phone calls or text messages; she can also run more advanced virus removal tools to clean the mobile device. Hence, VirusMeter is a valuable tool to expose malware on mobile devices to their users at their early stages, thus preventing them from continuously compromising the service security or data confidentiality of the mobile device.

The pre-defined power model in VirusMeter is user-centric, which is relative to system-centric models that typically have too many system-level states. This model is constructed by VirusMeter itself when the device is in a clean state. VirusMeter consists of three major components: *User-Centric Power Model*,

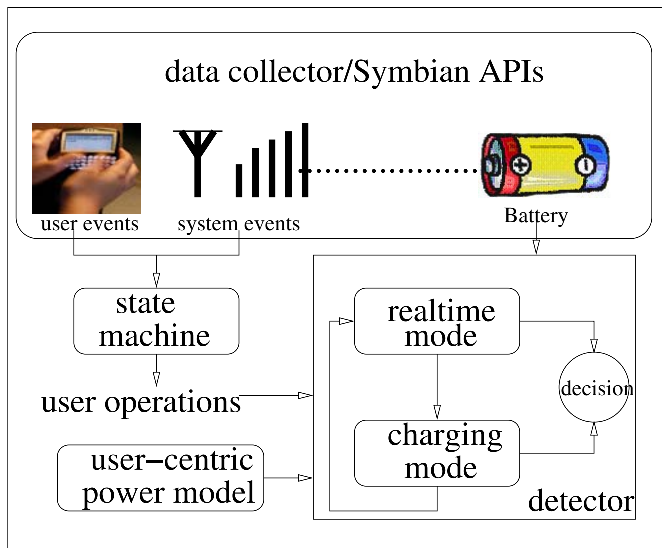


Fig. 2. VirusMeter Architecture

Data Collector, and *Malware Detector*. Figure 2 shows these components and the work flow of VirusMeter.

While the logic of VirusMeter is straightforward, we still need to overcome several challenges before VirusMeter becomes practically deployable on commodity mobile devices: **(1) Accurate power modeling:** An accurate yet simple power consumption model is crucial to the effectiveness of VirusMeter for malware detection. **(2) Precise power measurement:** Both model construction and data collection rely on precise power measurement. **(3) Low execution overhead:** For VirusMeter to be practically deployable, its own power consumption should not adversely affect the power-based anomaly detection.

In the following sections, we shall present more design and implementation details that address these challenges.

4 Building a User-Centric Power Model for VirusMeter

4.1 Existing Battery Power Models

Generally speaking, a battery's power consumption rate can be affected by two groups of factors, *environmental factors* such as signal strength, environmental noises, temperature, humidity, the distance to the base station, the discharging rate, the remaining battery power, etc., and *user operations* such as phone calls, emailing, text messaging, music playing, etc. Three types of power models have been suggested so far:

(1) Linear Model: In this simple model the remaining capacity after operating duration t_d is given by

$$P_r = P_p - \int_{t=t_0}^{t_0+t_d} d(t)dt = P_p - I \times t_d, \quad (1)$$

where P_p is the previous battery power, and $d(t)$ is the draining rate at time t. With the assumption that the operating mode does not change for t_d time units, $d(t)$ stays the same during this period and is denoted as I. Once the operation mode changes, the remaining capacity is re-calculated [28].

(2) Discharge Rate Dependent Model: In this model, the discharge rate is considered to be related to the battery capacity. For this purpose, c is defined as the fraction of the effective battery capacity P_{eff} and the maximum capacity P_{max}, i.e., $c = \frac{P_{eff}}{P_{max}}$. Then the battery power is calculated as

$$P_r = c \times P_p - \int_{t=t_0}^{t_0+t_d} d(t)dt = c \times P_p - I \times t_d. \quad (2)$$

c changes with the current; it becomes close to 1 when the discharge rate is low, and approaches 0 when the discharge rate is high [6,28].

(3) Relaxation Model: This model is based on a common phenomenon called relaxation [14,18], which refers to the fact that when a battery is discharged at a high rate, the diffusion rate of the active ingredients through the electrolyte and electrode will fall behind, and the battery reaches its end of life even if there are active materials available. If the discharge current is cut off or reduced, the diffusion and transport rate of active materials will catch up with the depletion of the materials [25]. Although this is the most comprehensive model characterizing a real battery, the model involves more than 50 electro-chemical and physical input parameters [25].

All these models calculate the battery power consumption from a physical and electrical perspective, although their inputs are remarkably different. The relaxation model can provide more accurate battery estimation than the linear model. However, even with aid of external instruments, measuring over 50 parameters could be difficult and expensive in practice. In addition, since VirusMeter aims to run on commodity mobile devices, it purely relies on publicly available system functions (without external support) to collect data; most of the 50 parameters in the relaxation model, however, cannot be captured with available APIs. Furthermore, a model with as many as 50 parameters is too cumbersome and thus not suitable for resource-constrained devices. The other two models model have similar problems, as the power draining rate and discharge rate are hard to measure without external power measurement instruments.

4.2 User-Centric Power Model

Due to the difficulties of measuring the input parameters of existing power models, we decide to build a user-centric power model for VirusMeter. In this model, the amount of power consumed is characterized as a function of common user operations and relevant environmental factors. Moreover, this model has only a

few states, which is in contrast to those system-centric power models that need cumbersomely profile all system behaviors and are thus difficult to build without in-depth understanding of the mobile OS and its underlying hardware.

To derive a user-centric model from scratch, we investigate the power consumption of common types of user operations on mobile devices in different environments. The following types of user operations are now considered: *(1) Calling*: its power consumption is mainly dependent on the conversation duration. VirusMeter treats incoming and outgoing calls separately. *(2) Messaging*: its average power consumption depends on both the sizes and the types of the messages. MMS and SMS are the two message types being considered. Also, sending and receiving messages are treated as different activities. *(3) Emailing*: its power consumption is mainly decided by the amount of traffic, which we can get by querying the email message size. *(4) Document processing*: we assume that the duration of the operation is the deciding factor. *(5) Web surfing*: Web surfing is more complicated than the above as a user may view, download, or be idle when surfing the Web. Currently we calculate the average power consumption simply based on the amount of traffic involved and also the surfing duration. *(6) Idle*: for a large amount of time, a user may not operate on the device for anything. During this period, however, system activities such as signaling may still take place. Under such a state, the power consumption is intuitively relevant to its duration. *(7) Entertainment and others*: currently, we simply assume the average power consumption is determined by the duration of the activities. This, admittedly, is a coarse model and further study is required.

For environmental factors, the following two types are being considered: *(1) Signal strength*: signal strength impacts the power consumption of all the above operations. The weaker of the signal strength, the more power consumption is expected. *(2) Network condition*: for some of the operations, network conditions are also important. For example, the time, and thus the power, needed to send a text message depends on the current network condition.

In VirusMeter, the battery power consumed between two measurements can be described as a function of all these factors during this period:

$$\Delta P = f(D^i_{call}, SS^i_{call}, T^j_{msg}, S^j_{msg}, SS^j_{msg}, N^j_{msg} ..., D^k_{idle}, SS^k_{idle}), \quad (3)$$

where ΔP represents the power consumption, D the duration of the operation, SS the signal strength, T the type of the text message, and N the network condition. i,j, and k represent the index of the user operation under discussion.

To this end, what is missing in this user-centric power model is the function itself in Equation 3. This is derived from the following three different approaches:

Linear Regression: Linear regression generates a mathematical function which linearly combines all variables we have discussed with techniques such as least square functions; it can thus be easily stored and implemented in a small segment of codes that run on commodity mobile devices with trivial overhead. While linear regression may incur little overhead, which makes it suitable for real-time detection, its accuracy depends on the underlying assumption of the linear relationship between variables.

Neural Network: An artificial neural network (ANN), often referred to as a "neural network" (NN), is a mathematical or computational model inspired by biological neural networks. It consists of an interconnected group of artificial neurons that process information using a connectionist approach for computation. Neural networks are commonly used for non-linear statistical data modeling. They can be used to model complex relationships between inputs and outputs or to find patterns in data. In VirusMeter, we use neural network as a regression tool, in which the neural network model, unlike the linear regression model, cannot easily be presented as a mathematical function.

Decision Trees: A decision tree is a predictive model that maps the observations of an item to conclusions of its target value. In a decision tree, branches represent conjunctions of features that lead to leaves that represent classifications. In VirusMeter we build a classification tree in which branches represent normal or malware samples. We train the decision tree with both normal and malware data samples. When a new piece of data sample is fed into the decision tree, it can tell if the new data is normal or not, as well as which malware most likely caused the abnormal power consumption.

5 Constructing State Machines for Data Collection

To train the three power models presented in the previous section, VirusMeter needs to collect some data. For the linear and neural network model construction, only clean data are needed. For decision tree construction, both clean data and dirty data (the data when malware programs are present) are needed. In this section, we present how VirusMeter collects these data to train the models.

Currently, we mainly consider the user operations defined in the previous section and their corresponding power consumption in VirusMeter. Although the power consumption can be queried using public APIs, there is no interface that could be directly called for the user operations. As it is common for commodity devices to provide some APIs for third parties to query, register, and monitor system-level events or status, we construct a state machine to derive user operations (which we also call *external* events) from system events (which we also call *internal* events). In this state machine, state transitions are triggered by internal events when they appear in a certain order and satisfy certain timing constraints. For example, during a normal incoming call, a ring event must precede another answer key event, but cannot happen more than 25 seconds before the answer key event, because ringing lasts for less than 25 seconds in our experimental cellphone before the call is forwarded to the voicemail service.

One may wonder whether we can simply use these state machines to detect malware without power auditing. This is possible but can potentially miss some malware for two reasons. On one hand, stealthy malware can easily evade detection by mimicing normal user behaviors that can be derived from the state machine. On the other hand, it is difficult, if not impossible, to build a state machine that exhaustively characterizes all possible user operations. The state machine in VirusMeter covers only the internal events corresponding to those

Algorithm 1. State Machine Construction for Each User Operation

1: Run a monitor program on the clean cellphone.
2: Execute a defined user operation, such as a phone call.
3: Monitor and record all related internal events during the test period and their properties.
4: Find the correlation between a user operation and the internal events, their dependency and sequences.
5: Query and record all parameters of the events.
6: Repeat the experiment.
7: Abstract the common event sequence from the recording. These internal events are used to build the state machine.

common user operations that we have defined. Due to these concerns, we still need the power model for mobile malware detection.

VirusMeter performs Algorithm 1 to construct the state machine for each user operation defined previously. Figure 3 shows an example of the obtained state machine for receiving a phone call. In this figure, the triggering events are marked on the transition arrows. Starting in the `Idle` state, the state machine transits to the `Ring` state after a `ring` event. If the user decides to answer the call by pressing the answer key, the answer key event is generated, which makes the state machine move to the `Answer` state if the answer key event happens half a second to 25 seconds after the `Ring` state. On a Symbian cell phone, we can observe an *EStatusAnswering* event. At this time, the state machine starts a timer. When the user terminates the call by pressing the cancel key or hanging it up, the state machine turns to the `End` state followed by a Symbian *EStatusDisconnecting* event. The state machine now stops the timer and calculates the calling duration. Finally the state machine returns to `Idle` state and generates a `receiving call` operation with the call duration. In a similar approach, we conduct experiments to build state machines for other user operations we have defined.

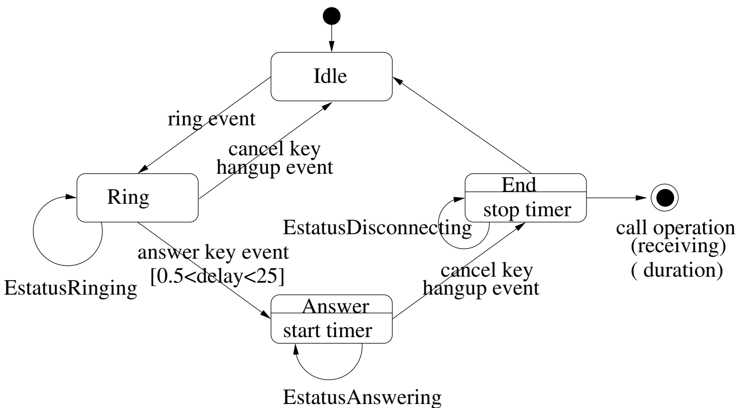


Fig. 3. State Machine for Receiving a Phone Call

6 Model Checking for Malware Detection

With the power model and the state machines available, VirusMeter can perform malware detection in a straightforward manner: we use the power model to predict how much power should be consumed and then compare it against the measured power consumption. If abnormal power consumption is observed, an alert is raised. Here, VirusMeter is designed with two running modes:

- *Real-time mode*: VirusMeter uses the linear regression power model to predict power consumption due to its low computational cost.
- *Battery-charging mode*: Although linear regression is easy to perform, it may generate false detection results since (1) it implicitly assumes a linear relationship among all variables, and (2) power measurements may have fluctuations due to electro-chemical battery properties. Thus, VirusMeter accumulates power consumption measurement data and uses the neural network model and the decision tree algorithm to perform malware detection when the battery is charging.

It is noted that both modes can also run off the mobile device. For example, the device manufacturer or the service operator may provide such a service that a user can submit the collected measurement data to a server for malware detection. This however will increase the communication cost on the mobile device.

7 Practical Issues in VirusMeter Implementation

As Symbian is the most popular mobile OS, we implement a prototype of VirusMeter on Nokia 5500 Sport, supported by Symbian OS 9.1. Figure 4 shows the modules of VirusMeter implementation. Currently, it is implemented as a user-level application and a user can choose to start it or to shut it down manually. The implementation program uses a client/server architecture which is widely used for Symbian applications. Figure 5 shows the user interface once VirusMeter is installed on our experimental device.

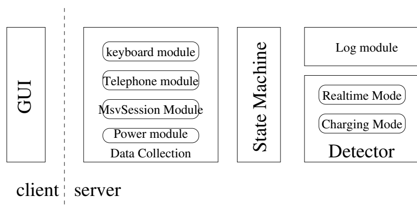


Fig. 4. Modularized VirusMeter Implementation

Fig. 5. VirusMeter Implemented on a Nokia 5500 Sport

7.1 Power Measurement Precision and Power Model Construction

The power consumption data are collected through the APIs provided by Symbian for power status changes. In the prototype implementation, however, we find that the precision of the power capacity measurement is not sufficient. In fact, the precision returned by the APIs of mobile devices varies significantly. For example, iPhone can return the current power capacity at 1% precision. Many other devices, including the one in our experiments, return the power consumption data only at the level of battery bars shown on the screen. On the Nokia 5500, these bars are at the 100, 85, 71, 57, 42, 28, 14, and 0 percent of the full capacity. We call the battery supply between two of these successive values as a power *segment*. To overcome the precision challenge, we perform experiments long enough so that the power consumption is sufficient to cross a segment. Assuming a constant draining rate during the experiments, we expect the power measurement through this method is more accurate.

Accordingly, it is necessary to transform the power model in Equation 3 because if we were to still follow Equation 3, many experiment samples would have the same constant dependent value ΔP, which is bad for the linear regression and neural network regression. To make the regression as accurately as possible, we transform the function as follows. Because in all our experiments, the signal strength is always good (at level 6 and 7) but the duration of idle time has a large range, we select idle time at the best signal strength as the dependent variable, and transform our model to

$$D_{idle} = f'(D^i_{call}, SS^i_{call}, T^j_{msg}, S^j_{msg}, SS^j_{msg}, N^j_{msg}..., \Delta P, SS^k_{idle}). \qquad (4)$$

For environmental factors, VirusMeter is currently only concerned about the signal strength and network condition. Through the API, VirusMeter can directly query the current signal strength. There are 7 levels of signal strength on Nokia 5500, from 1 to 7. We, however, cannot directly query APIs for network conditions when a user performs a certain operation, such as text messaging. In the experiments, we have observed that if the network congestion is severe, the duration for sending or receiving messages increase significantly. Therefore, to make the power model more accurate, we introduce the sending time into it, and the duration is measured as follows. In Symbian, sending a message leads to a sequence of events that can be captured by VirusMeter: first, an `index` is created in the `draft` directory; when the creation is complete, the `index` is moved to the `sending` directory; when sending is successful, the `index` will be moved to the `sent` directory. Hence, the operation time can be measured from the time when the `index` is created to the time when it is moved to the `sent` directory. Following the similar idea, we further refine the parameter input for receiving messages and other networking operations.

Note that our power model is built in such a way due to insufficient power precision, but a malware does not need to be active throughout a segment of battery power to be detected by VirusMeter. Instead, no matter how long the malware is active, we can always feed the runtime data collected during an entire

power segment for malware detection, and our experiments in the next section will show that it is still very effective.

7.2 Data Collection Rules

To construct the power model, we need to collect not only the power consumption data under normal user operations (clean data) for the three power models, but also dirty data when malware is present for training the decision tree. Constrained by the precision of the battery power measurement offered by Symbian OS, we treat all user operations conducted in one battery segment as a batch to achieve more accurate detection. As our goal is to detect malware whose activities lead to abnormal power consumption no matter how long they are active, we collect clean data under various circumstances for model construction: (1) In some experiments, our data collection just focuses on a single user operation. For example, in a battery segment, we only send SMS text messages, and in another one, we only receive SMS text messages; (2) In some experiments, mixed user operations are conducted. For example, in a battery segment, we make phone calls and also receive text messages; (3) For each user operation, we consider various properties of the activity. For instance, we send text messages with different sizes ranging from ten bytes to a thousand bytes; and (4) In all experiments, we avoid abnormal conditions, which decrease the accuracy of our power models.

Dirty data are also necessary to train the decision trees. The power consumption of a malware program may vary significantly in different environments. For example, different usage frequencies or spy call durations on FlexiSPY cause great difference in power consumption. In another example, the power consumed by the Cabir worm depends on how many Bluetooth devices exist in the neighborhood. Based on such considerations, we collect dirty data as follows: (1) During dirty data collection, we conduct experiments to cover as many different scenarios as possible, including both high power consumption cases and low power consumption cases; and (2) For the purpose of model training, the fraction of high and low power consumption data samples are randomly selected.

7.3 Stepwise Regression for Data Pre-processing and Time-Series Data Analysis

The data we have collected, including both clean and dirty data, have 41 variables that are measurable through the Symbian APIs. To simplify the model by eliminating insignificant factors, we first use the `stepwise regression` technique [7] to pre-process the collected data. Stepwise regression is a statistical tool that helps find the most significant terms and remove least significant ones. Besides that, stepwise regression also provides information that help to merge variables. Using stepwise regression, we found that the idle time with signal strength level 6 is insignificant. This is because in our experimental environment, we often have good signal strength at level 7. The signal strength 6 is relatively rare. Thus, we merge the signal strength 6 to the signal strength 7.

To further improve the model accuracy, we collect data samples from multiple segments and use the average to smooth out the fluctuations due to the internal

electro-chemical battery properties. Based on this idea, we generate three sets of input for each power model. If a model is built from data samples collected in a single battery power segment, we call them "short-term" experiments. If a model is built from data samples from seven segments, we call them "middle-term" experiments. Note that Nokia 5500 only has seven battery segments. We can further feed data samples collected in more than one battery lifecycle. In our experiments, we use four battery lifecycles, which correspond to 28 segments, and we call them "long-term" experiments. A stealthy malware that does not consume much power in one segment may not be caught in a short-term detection, but can be caught in the middle- or long-term detection.

8 Evaluation Results

In this section, we use actual mobile malware, including FlexiSPY, Cabir, and some variants of Cabir, to evaluate the effectiveness of VirusMeter. `FlexiSPY` is a spyware program that runs on either Symbian OS or Blackberry handhelds. Once installed, it conducts eavesdropping, call interception, GPS tracking, etc. It monitors phone calls and SMS text messages and can be configured to send them to a remote server. We test three major types of misbehaviors supported by FlexiSPY: *eavesdropping (spy call)*, *call interception*, and *message (text message and email) forwarding*. Figure 6 shows the information flow of FlexiSPY. The `Cabir` malware exploit Bluetooth to spread themselves. We obtained 3 Cabir variants and in the experiments, we used two of them for decision tree training and the other one for testing.

We have several sets of experiments to examine common malware behaviors that consume low (such as Cabir), medium (such as text-message forwarding), and high battery power (such as eavesdropping). We also evaluate false positives and the runtime overhead, i.e., power consumption, of VirusMeter.

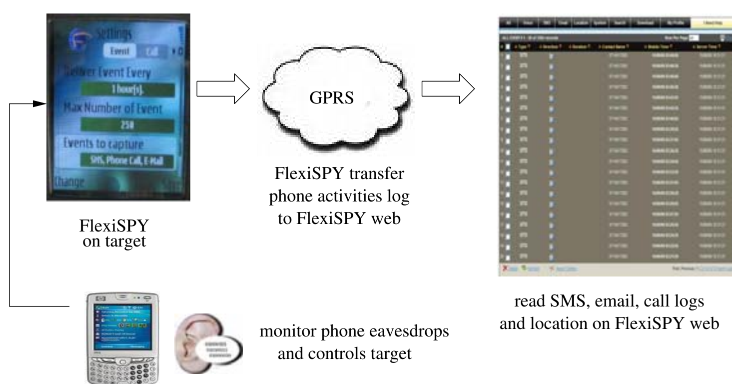


Fig. 6. FlexiSPY Running on Nokia 5500 Sport and the Information Flow

Table 1. Detection Rate (%) on Eavesdropping

	Short-Term	Middle-Term	Long-Term
Linear Regression	85.1	89.9	87.1
Neural Network	89.3	90.9	93.0
Decision Tree	89.8	90.2	88.9

8.1 Experiments on Eavesdropping Detection

When using FlexiSPY to eavesdrop on a cellphone, the attacker makes a call to a previously configured phone number and then the phone is activated silently without user authentication. Our power measurement shows that eavesdropping has a similar power consumption rate as a normal call. In our experiments, we make spy calls of different time durations uniformly ranging from 1 minute to 30 minutes. More than 50 samples are collected in this and each of the following detection rate experiments. Table 1 shows the detection rates (true positives).

The results show that for eavesdropping, both middle-term and long-term experiments can improve the detection rates for linear regression and neural network, compared with short-term detection. In fact, even the short-term linear regression achieves a detection rate over 85%. This is because eavesdropping consumes a lot of power, which makes short-term detection quite accurate. Surprisingly, the long-term detection based on linear regression generates a worse result than mid-term detection. Our conjecture is that due to the inaccurate linear relationship between variables, more errors may be accumulated in the long-term experiments, which leads to worse results. This may apply to long-term decision tree as well.

8.2 Experiments on Call Interception Detection

FlexiSPY can also perform call interceptions, which enables the attacker to monitor ongoing calls. A call interception differs from eavesdropping in that the call interception can only be conducted when a call is active. After FlexiSPY is installed, when the victim makes a call to a pre-set phone number, the attacker will automatically receive a notification via text message and silently call the victim to begin the interception.

In our detection experiments, we again perform call interceptions with different time durations uniformly ranging from 1 minute to 30 minutes. Table 2 shows the detection rate. The short-term linear regression detection results are

Table 2. Detection Rate (%) on Call Interception

	Short-Term	Middle-Term	Long-Term
Linear Regression	66.8	79.5	82.4
Neural Network	82.9	86.0	90.5
Decision Tree	84.8	86.8	86.9

Table 3. Detection Rate (%) on Text Message Forwarding

	Short-Term	Middle-Term	Long-Term
Linear Regression	89.5	93.0	96.4
Neural Network	90.3	94.8	98.6
Decision Tree	88.7	89.1	90.7

not very good when compared to neural network and decision tree. This is because the call interception only consumes slightly more battery power than a normal phone call and it only works when a call is active. But middle-term and long-term experiments can significantly improve the detection rate for linear regression. The results confirm that for stealthy malware that consumes only a small amount of power, a more accurate model or a longer detection time can help improve the detection accuracy.

8.3 Experiments on Text-Message Forwarding and Information Leaking Detection

FlexiSPY can also collect user events, such as call logs, and then deliver collected information via a GPRS connection periodically at a pre-configured time interval. Clearly, transferring data through GPRS consumes power and the power consumption depends on the time interval and the characteristics of user operations such as the number of text messages sent during each interval.

In our detection experiments, we set the interval from 30 minutes to 6 hours, with an interval of 30 minutes. Under each setting, we keep sending and receiving text messages of different sizes ranging from 10 bytes to 1000 bytes.Table 3 shows the detection results. We can see all three approaches achieve detection rates above 88%. The long-term detection with linear regression and neural network can achieve a detection rate up to 98.6%. Our analysis shows that this is because such a FlexiSPY functionality consumes additional power other than communication: although when the interval is large, FlexiSPY may not transfer data for a while, FlexiSPY still needs to monitor and save information related to user activities, which also consumes battery power. Thus, even in short-term experiments, the detection rate is quite high. To our surprise, decision tree does not achieve comparable results to linear regression and neural networks for middle-term and long-term detection. We believe that the performance of decision tree is highly related to the training dataset, for which we are currently constrained by a limited number of malware samples.

8.4 Experiments on Detecting Cabir

Cabir, a cellphone worm spreading via Bluetooth, searches nearby Bluetooth equipments and then transfers a `sis` file to them once found. The power consumption of Cabir mainly comes from two parts: neighbor discovery and file transferring. Because Bluetooth normally does not consume significant battery power, we conduct the experiments in an environment full of Bluetooth

Table 4. Detection Rate (%) on Cabir

	Short-Term	Middle-Term	Long-Term
Linear Regression	84.6	89.8	92.9
Neural Network	88.6	93.4	93.5
Decision Tree	86.8	87.6	88.7

Table 5. Detection Rate (%) on Multiple Malware Infection

	Short-Term	Middle-Term	Long-Term
Linear Regression	84.8	87.9	88.1
Neural Network	88.9	90.2	92.0
Decision Tree	72.6	76.3	73.6

equipments, in which Cabir keeps finding new equipments and thus consumes a nontrivial amount of power. To control the frequency of file transferring, we repeatedly turn off Bluetooth on these devices for a random amount of time after a transfer completes and then turns it on again.

Table 4 shows the detection results. For linear regression, the middle-term and long-term detection can remarkably improve the detection result. The table also indicates that although Bluetooth discovery and file transferring only consume a limited amount of battery power, it can be detected with a reasonably high rate by VirusMeter at real time.

8.5 Experiments on Detecting Multiple Malware Infections

Previous detection experiments all involve only one malware program running on the cellphone. It is possible that a mobile device is infected by more than one malware program and each malware program could perform different attacks simultaneously. To test such cases, we activate both FlexiSPY and Cabir on our experimental cellphone and randomly conduct various attack combinations.

Table 5 shows the detection rates. The results show that both linear regression and neural network still have reasonably high true positive rates. But decision tree results in a much higher false negative rate than in single malware infection experiments. Although it is seemingly counterintuitive, the underlying principle of these three approaches can explain this: linear regression and neural network regression only predict the power consumption of normal user operations rather than describing power consumption of specific malware activities, which is the objective of decision tree. However, our decision tree model is not trained with a mixture of malware samples. Thus for data samples collected when multiple malware programs are active, its performance is the worst.

Table 6. VirusMeter False Positive Rate (%)

	Short-Term	Middle-Term	Long-Term
Linear Regression	22.4	14.2	10.3
Neural Network	10.0	5.1	4.3
Decision Tree	15.2	15.1	14.4

8.6 False Positive Experiments

In addition to the detection rates, we also conduct experiments to evaluate false positives. By feeding power models with a clean dataset, we can get the prediction result and calculate the false positive rate. For this purpose, we collect more than 100 clean data samples for experiments.

Table 6 shows the false positive rates. The results show that linear regression in short-term detection has the highest false positive rate. This is due to the inaccuracy of the underlying assumption of linear regression model. However, both middle-term and long-term experiments can significantly reduce the false positive rates. With a more accurate power model, neural network achieves the best results among the three for all three terms.

8.7 VirusMeter Overhead Measurement

As VirusMeter targets to run on commodity devices, its power consumption overhead is a great concern. As we cannot directly measure the power consumption of VirusMeter, we conduct experiments as follows: with and without VirusMeter running on the cellphone, we conduct the same set of user operations and then

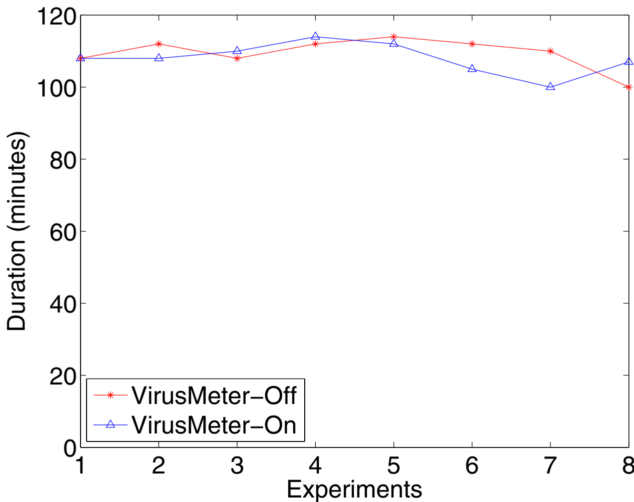


Fig. 7. VirusMeter overhead evaluation

compare the operating durations under these two scenarios. Figure 7 shows our experimental results. We can see that with and without VirusMeter, the duration of various user operations is very close. The average duration when VirusMeter is off is 109.5 minutes across our experiments, while the average duration when VirusMeter is on is 108 minutes. This indicates the VirusMeter running overhead is less than 1.5%. Note the above results show the overhead when VirusMeter uses the linear regression model. For the other two approaches, we do not evaluate their power consumption because they run in a battery-charging mode.

9 Further Discussion

VirusMeter, in principle, has the potential to detect any misbehavior with abnormal power consumption as long as the battery power metering is sufficiently accurate. Currently, precisions of battery power indicators vary significantly among different mobile OSes, which can affect the detection efficiency of VirusMeter. This is particularly important for real-time detection. Practically, on our experimental device, this changes the real-time detection mode of VirusMeter to a near-real-time mode.

Since VirusMeter relies on the user-centric power models to detect malware, the accuracy of the models themselves is important. Our experimental results have shown that linear regression, although consuming trivial additional power, may generate high false negative rates due to the inaccurate underlying assumption between variables. On the other hand, in a battery-charging mode, neural network often improves the detection rate remarkably due to lack of such an assumption. The decision tree model does not perform as effectively as neural networks in our experiments. We believe that our limited malware samples may adversely affect its performance, and we plan to collect more samples in the future to further evaluate this method. In addition, for some types of user operations, such as entertainment and Web surfing, more fine-grained profiling can further improve the accuracy of the power model.

As we suggested, VirusMeter can also run in the battery-charging mode to improve the detection accuracy. Malware may leverage this as well since when the battery is charging, there is no way for VirusMeter to accurately measure the power consumption without any external assistance. To capture this kind of malware, VirusMeter needs external devices to measure how much power is charged and how much power is consumed. On the other hand, currently most mobile OSes are only accessible to manufacturers. If we place VirusMeter in the mobile OS, VirusMeter becomes more resilient to those attacks that could fail signature- or anomaly-based detection schemes. But if the mobile OS is also cracked and the malware knows how to inject faked events, VirusMeter will also fail because the data collected by VirusMeter cannot be trusted any more.

10 Conclusion

The battery power supply is often regarded as the Achilles' heel of mobile devices. Provided that any activity conducted on a mobile device, either normal or

malicious, inevitably consumes some battery power, VirusMeter exploits this to detect existence of malware with abnormal power consumption. VirusMeter relies on a concise lightweight user-centric power model and aims to detect mobile malware in two modes: While the real-time detection mode provides immediate detection, running VirusMeter under the battery-charging mode can further improve the detection accuracy without concerns of resource consumption. Using real-world malware such as Cabir and FlexiSpy, we experimentally show that VirusMeter can effectively and efficiently detect their existence.

Acknowledgment

We thank the anonymous referees for providing constructive comments. The work has been supported in part by U.S. AFOSR under grant FA9550-09-1-0071, and by U.S. National Science Foundation under grants CNS-0509061, CNS-0621631, and CNS-0746649.

References

1. http://www.wellingtonfund.com/blog/2007/02/19/gmp-3gsm-wrapup/
2. http://en.wikipedia.org/wiki/Smartphone
3. http://www.viruslibrary.com/
4. http://vx.netlux.org/29a/
5. http://www.flexispy.com/
6. http://www.panasonic.com/inustrial_oem/battery/battery_oem/chem/lith/lith.htm
7. http://en.wikipedia.org/wiki/Stepwise_regression
8. Commwarrior, http://www.f-secure.com/v-descs/commwarrior.shtml
9. Sprots fans in helsinki falling prey to cabir, http://news.zdnet.com
10. Bose, A., Hu, X., Shin, K., Park, T.: Behavioral detection of malware on mobile handsets. In: Proceedings of Mobisys, Breckenridge, CO (June 2008)
11. Bose, A., Shin, K.: On mobile virus exploiting messaging and bluetooth services. In: Proceedings of Securecomm (2006)
12. Bose, A., Shin, K.: Proactive security for mobile messaging networks. In: Proceedings of WiSe (2006)
13. Cheng, J., Wong, S., Yang, H., Lu, S.: Smartsiren: Virus detection and alert for smartphones. In: Proceedings of ACM MobiSys, San Juan, Puerto Rico (2007)
14. Chiasserini, C., Rao, R.: Pulsed battery discharge in communication devices. In: Proceedings of MobiComm, Seattle, WA (August 1999)
15. Dagon, D., Martin, T., Starner, T.: Mobile phones as computing devices: The viruses are coming! IEEE Pervasive Computing (2004)
16. Enck, W., Traynor, P., McDaniel, P., Porta, T.: Exploiting open functionality in sms-capable cellular networks. In: Proceedings of CCS 2005 (November 2005)
17. Fleizach, C., Liljenstam, M., Johansson, P., Voelker, G., Mehes, A.: Can you infect me now? malware propagation in mobile phone networks. In: Proceedings of WORMS, Alexandria, VA (November 2007)
18. Fuller, T., Doyle, M., Newman, J.: Simulation and optimization of the dual lithium ion insertion cell. Journal of Electrochem. Soc. 141 (April 1994)

19. Guo, C., Wang, H., Zhu, W.: Smart-phone attacks and defenses. In: Proceedings of HotNets III, San Diego, CA (November 2004)
20. Hu, G., Venugopal, D.: A malware signature extraction and detection method applied to mobile networks. In: Proceedings of IPCCC (April 2007)
21. Hypponen, M.: http://www.usenix.org/events/sec07/tech/hypponen.pdf
22. Kim, H., Smith, J., Shin, K.: Detecting energy-greedy anomalies and mobile malware variants. In: Proceedings of Mobisys, Breckenridge, CO (June 2008)
23. Mickens, J., Noble, B.: Modeling epidemic spreading in mobile networks. In: Proceedings of ACM WiSe (2005)
24. Mulliner, C., Vigna, G., Dagon, D., Lee, W.: Using labeling to prevent cross-service attacks against smart phones. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 91–108. Springer, Heidelberg (2006)
25. Park, S., Savvides, A., Srivastava, M.: Battery capacity measurement and analysis using lithium coin cell battery. In: Proceedings of ISLPED (August 2001)
26. Racic, R., Ma, D., Chen, H.: Exploiting mms vulnerabilities to stealthily exhaust mobile phone's battery. In: Proceedings of SecureComm 2006 (August 2006)
27. Sarat, S., Terzis, A.: On the detection and origin identification of mobile worms. In: Proceedings of WORMS, Alexandria, VA (November 2007)
28. Simunic, T., Benini, L., Micheli, G.: Energy-efficient design of battery-powered embedded systems. In: Proceedings of ISLPED (August 1999)
29. Su, J., Chan, K., Miklas, A., Po, K., Akhavan, A., Saroiu, S., Lara, E., Goel, A.: A preliminary investigation of worm infections in a bluetooth environment. In: Proceedings of WORM (2006)
30. Traynor, P., Enck, W., McDaniel, P., Porta, T.: Mitigating attacks on open functionality in sms-capable cellular networks. In: Proceedings of Mobicom 2006 (2006)
31. Venugopal, D., Hu, G., Roman, N.: Intelligent virus detection on mobile devices. In: Proceedings of ACM PST, Markham, Ontario, Canada (October 2006)
32. Yan, G., Eidenbenz, S.: Modeling propagation dynamics of bluetooth worms. In: Proceedings of ICDCS 2007 (2007)

Regular Expression Matching on Graphics Hardware for Intrusion Detection

Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis

Institute of Computer Science, Foundation for Research and Technology – Hellas
{gvasil,mikepo,antonat,markatos,sotiris}@ics.forth.gr

Abstract. The expressive power of regular expressions has been often exploited in network intrusion detection systems, virus scanners, and spam filtering applications. However, the flexible pattern matching functionality of regular expressions in these systems comes with significant overheads in terms of both memory and CPU cycles, since every byte of the inspected input needs to be processed and compared against a large set of regular expressions.

In this paper we present the design, implementation and evaluation of a *regular expression matching engine* running on graphics processing units (GPUs). The significant spare computational power and data parallelism capabilities of modern GPUs permits the efficient matching of multiple inputs at the same time against a large set of regular expressions. Our evaluation shows that regular expression matching on graphics hardware can result to a 48 times speedup over traditional CPU implementations and up to 16 Gbit/s in processing throughput. We demonstrate the feasibility of GPU regular expression matching by implementing it in the popular Snort intrusion detection system, which results to a 60% increase in the packet processing throughput.

1 Introduction

Network Intrusion Detection Systems (NIDS) are an efficient mechanism for detecting and preventing well-known attacks. The typical use of a NIDS is to passively examine network traffic and detect intrusion attempts and other known threats. Most modern network intrusion detection and prevention systems rely on deep packet inspection to determine whether a packet contains an attack vector or not. Traditionally, deep packet inspection has been limited to directly comparing the packet payload against a set of string literals. One or more string literals combined into a single *rule* are used to describe a known attack. By using raw byte sequences extracted from the attack vector, it is easy to maintain signature sets that describe a large number of known threats and also make them easily accessible to the public.

However, the existence of *loose* signatures [28] can increase the number of false positives. Signatures that fail to precisely describe a given attack may increase the number of matches in traffic that do not contain an actual attack.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 265–283, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Moreover, string literals that are shared between two or more rules will probably conflict at the matching phase and increase the number of false positives. Thus, a large number of well and carefully designed strings may be required for precisely describing a known attack.

On the other hand, regular expressions are much more expressive and flexible than simple byte sequences, and therefore can describe a wider variety of payload signatures. A single regular expression can cover a large number of individual string representations, and thus regular expressions have become essential for representing threat signatures for intrusion detection systems. Several NIDSes, such as Snort [21] and Bro [20] contain a large number of regular expressions to accomplish more accurate results. Unfortunately, regular expression matching, is a highly computationally intensive process. This overhead is due to the fact that, most of the time, every byte of every packet needs to be processed as part of the detection algorithm that searches for matches among a large set of expressions from all signatures that apply to a particular packet.

A possible solution is the use of hardware platforms to perform regular expression matching [9,24,7,18]. Specialized devices, such as ASICs and FPGAs, can be used to inspect many packets concurrently. Both are very efficient and perform well, however they are complex to modify and program. Moreover, FPGA-based architectures have poor flexibility, since most of the approaches are usually tied to a specific implementation.

In contrast, commodity graphics processing units (GPUs) have been proven to be very efficient for accelerating the string searching operations of NIDS [14,30, 10]. Modern GPUs are specialized for computationally-intensive and highly parallel operations—mandatory for graphics rendering—and therefore are designed with more transistors devoted to data processing rather than data caching and flow control [19]. Moreover, the ever-growing video game industry exerts strong economic pressure for more powerful and flexible graphics processors.

In this paper we present the design, implementation, and evaluation of a GPU-based *regular expression matching engine* tailored to intrusion detection systems. We have extended the architecture of Gnort [30], which is based on the Snort IDS [21], such that *both pattern matching and regular expressions* are executed on the GPU. Our experimental results show that regular expression matching on graphics hardware can provide up to 48 times speedup over traditional CPU implementations and up to 16 Gbit/s of raw processing throughput. The computational throughput achieved by the graphics processor is worth the extra communication overhead needed to transfer network packets to the memory space of the GPU. We show that the overall processing throughput of Snort can be increased up to eight times compared to the default implementation.

The remainder of the paper is organized as follows. Background information on regular expressions and graphics processors is presented in Section 2. Section 3 describes our proposed architecture for matching regular expressions on a graphics processor, while Section 4 presents the details of our implementation in Snort. In Section 5 we evaluate our prototype system. The paper ends with an outline of related work in Section 7 and some concluding remarks in Section 8.

2 Background

In this section we briefly describe the architecture of modern graphics cards, and the general-purpose computing functionality they provide for non-graphics applications. We also discuss some general aspects of regular expression matching and how it is applied in network intrusion detection systems.

2.1 Graphics Processors

Graphics Processing Units (GPUs) have become powerful and ubiquitous. Besides accelerating graphics-intensive applications, vendors like NVIDIA[1] and ATI,[2] have started to promote the use of GPUs as general-purpose computational units complementary to the CPU.

In this work, we have chosen to work with the NVIDIA GeForce 9 Series (G9x) architecture, which offers a rich programming environment and flexible abstraction models through the Compute Unified Device Architecture (CUDA) SDK [19]. The CUDA programming model extends the C programming language with directives and libraries that abstract the underlying GPU architecture and make it more suitable for general purpose computing. CUDA also offers highly optimized data transfer operations to and from the GPU.

The G9x architecture, similarly to the previous G80 architecture, is based on a set of *multiprocessors*, each comprising a set of *stream processors* operating on SPMD (Single Process, Multiple Data) programs. A unit of work issued by the host computer to the GPU is called a *kernel* and is executed on the GPU as many different *threads* organized in *thread blocks*. A fast *shared memory* is managed explicitly by the programmer among thread blocks. The *global*, *constant*, and *texture memory spaces* can be read from or written to by the host, are persistent across kernel launched by the same application, and are optimized for different memory usage [19]. The constant and texture memory accesses are cached, so a read from them costs much less compared to device memory reads, which are not being cached. The texture memory space is implemented as a read-only region of device memory.

2.2 Regular Expressions

Regular expressions offer significant advantages over exact string matching, providing flexibility and expressiveness in specifying the context of each match. In particular, the use of logical operators is very useful for specifying the context for matching a relevant pattern. Regular expressions can be matched efficiently by compiling the expressions into state machines, in a similar way to some fixed string pattern matching algorithms [3].

A state machine can be either a deterministic (DFA) or non-deterministic (NFA) automaton, with each approach having its own advantages and disadvantages. An NFA can compactly represent multiple signatures but may result to long matching times, because the matching operation needs to explore multiple paths in the automaton in order to determine whether the input matches any signatures.

[1] http://developer.nvidia.com/object/cuda.html

[2] http://ati.amd.com/technology/streamcomputing/index.html

A DFA, on the other hand, can be efficiently implemented in software—a sequence of n bytes can be matched with $O(n)$ operations, which is very efficient in terms of speed. This is achieved because at any state, every possible input letter leads to at most one new state. An NFA in contrast, may have a set of alternative states to which it may backtrack when there is a mismatch on the previously selected path. However, DFAs usually require large amounts of memory to achieve this performance. In fact, complex regular expressions can exponentially increase the size of the resulting deterministic automaton [6].

2.3 Regular Expression Matching in Snort

Regular expression matching in Snort is implemented using the PCRE library [1]. The PCRE library uses an NFA structure by default, although it also supports DFA matching. PCRE provides a rich syntax for creating descriptive expressions, as well as extra modifiers that can enrich the behavior of the whole expression, such as case-insensitive or multi-line matching. In addition, Snort introduces its own modifiers based on internal information such as the position of the last pattern match, or the decoded URI. These modifiers are very useful in case an expression should be matched in relation to the end of the previous match.

Each regular expression is compiled into a separate automaton that is used at the searching phase to match the contents of a packet. Given the large number of regular expressions contained in Snort's default rule set, it would be inefficient to match every captured packet against each compiled automaton separately. 45% of the rules in the latest Snort ruleset perform regular expression matching, half of which are related to Web server protection.

To reduce the number of packets that need to be matched against a regular expression, Snort takes advantage of the string matching engine and uses it as a first-level filtering mechanism before proceeding to regular expression matching. Rules that contain a regular expression operation are augmented with a string searching operation that searches for the most characteristic fixed string counterpart of the regular expression used in the rule.

The string matching engine consists of a set-wise pattern matching algorithm that searches in advance for the fixed string subparts of all regular expressions simultaneously. For a given rule, if the fixed string parts of the regular expressions are not present in a packet, then the regular expression will never match. Thus, fixed string pattern matching acts as a pre-filtering mechanism to reduce the invocation of the regular expression matching engine, as shown in Figure 1.

There are also 24 rules in the latest Snort rule set that do not perform this pre-filtering, but we believe these are cases of poorly written rules. The matching procedure for regular expression matching is invoked only when the subparts have been identified in the packet. For example, in the following rule:

```
alert tcp any any -> any 21 (content:"PASS"; pcre:"/^PASS\s*\n/smi";)
```

the `pcre:` pattern will be evaluated only if the `content:` pattern has previously matched in the packet.

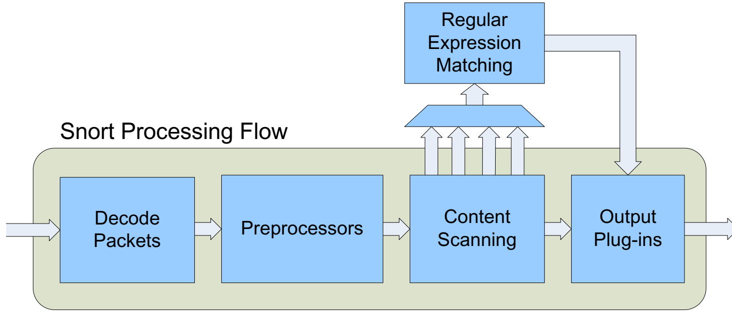


Fig. 1. Regular expression matching in the Snort IDS

3 Regular Expression Matching on Graphics Processors

We extend the architecture of Snort to make use of the GPU for offloading regular expression matching from the CPU, and decreasing its overall workload. Figure 2 depicts the top-level diagram of our regular expression pattern matching engine. Whenever a packet needs to be scanned against a regular expression, it is transferred to the GPU where the actual matching takes place. The SPMD operation of the GPU is ideal for creating multiple instantiations of regular expression state machines that will run on different stream processors and operate on different data.

Due to the overhead associated with a data transfer operation to the GPU, batching many small transfers into a larger one performs much better than making each transfer separately, as shown in Section 5.3. Thus, we have chosen to copy the packets to the GPU in batches. We use a separate buffer for temporarily storing the packets that need to be matched against a regular expression. Every time the buffer fills up, it is transferred to the GPU for execution.

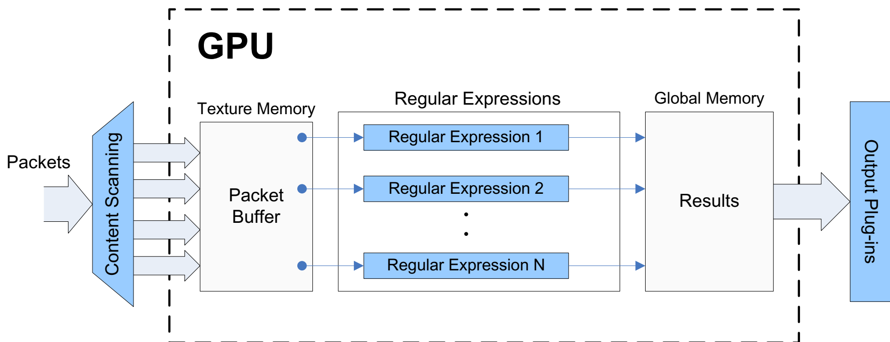


Fig. 2. Overview of the regular expression matching engine in the GPU

0 Reg.Ex. ID	4 Length	6 Payload 1536
Reg.Ex. ID	Length	Payload
Reg.Ex. ID	Length	Payload
Reg.Ex. ID	Length	Payload
⋮	⋮	⋮
Reg.Ex. ID	Length	Payload

Fig. 3. Packet buffer format

The content of the packet, as well as an identifier of the regular expression that needs to be matched against, are stored in the buffer as shown in Figure 3. Since each packet may need to be matched against a different expression, each packet is "marked" so that it can be processed by the appropriate regular expression at the search phase. Therefore, each row of the buffer contains a special field that is used to store a pointer to the state machine of the regular expression the specified packet should be scanned against.

Every time the buffer is filled up, it is processed by all the stream processors of the GPU at once. The matching process is a kernel function capable of scanning the payload of each network packet for a specific expression in parallel. The kernel function is executed simultaneously by many threads in parallel. Using the identifier of the regular expression, each thread will scan the whole packet in isolation. The state machines of all regular expressions are stored in the memory space of the graphics processor, thus they can be accessed directly by the stream processors and search the contents of the packets concurrently.

A major design decision for GPU regular expression matching is the type of automaton that will be used for the searching process. As we have discussed in Section 2, DFAs are far more efficient than the corresponding NFAs in terms of speed, thus we base our design of a DFA architecture capable of matching regular expressions on the GPU.

Given the rule set of Snort, all the contained regular expressions are compiled and converted into DFAs that are copied to the memory space of the GPU. The compilation process is performed by the CPU off-line at start-up. Each regular expression is compiled into a separate state machine table that is transferred to the memory space of the GPU. During the searching phase, all state machine tables reside in GPU memory only.

Our regular expression implementation currently does not support a few PCRE keywords related to some look-around expressions and back references. Back references use information about previously captured sub-patterns which is not straightforward to keep track of during searching. Look-around expressions scan the input data without consuming characters. In the current Snort default rule set, less than 2% of the rules that use regular expressions make use of these features. Therefore our regular expression compiler is able to generate automata for the vast majority of the regular expressions that are currently contained in

the Snort rule set. To preserve both accuracy and precision in attack detection, we use a hybrid approach in which all regular expressions that fail to compile into DFAs are matched on the CPU using a corresponding NFA, in the same way unmodified Snort does.

4 Implementation

In this section, we present the details of our implementation, which is based on the NVIDIA G9X platform using the CUDA programming model. First, we describe how the gathered network packets are collected and transferred to the memory space of the GPU. The GPU is not able to directly access the captured packets from the network interface, thus the packets must be copied by the CPU. Next, we describe how regular expressions are compiled and used directly by the graphics processor for efficiently inspecting the incoming data stream.

4.1 Collecting Packets on the CPU

An important performance factor of our architecture is the data transfers to and from the GPU. For that purpose, we use page-locked memory, which is substantially faster than non-page-locked memory, since it can be accessed directly by the GPU through Direct Memory Access (DMA). A limitation of this approach is that page locked memory is of limited size as it cannot be swapped. In practice though this is not a problem since modern PCs can be equipped with ample amounts of physical memory.

Having allocated a buffer for collecting the packets in page-locked memory, every time a packet is classified to be matched against a specific regular expression, it is copied to that buffer and is "marked" for searching against the corresponding finite automaton. We use a double-buffer scheme to permit overlap of computation and communication during data transfers between the GPU and CPU. Whenever the first buffer is transferred to the GPU through DMA, newly arriving packets are copied to the second buffer and vice versa.

A slight complication that must be handles comes from the TCP stream reassembly functionality of modern NIDSs, which reassembles distinct packets into TCP streams to prevent an attacker from evading detection by splitting the attack vector across multiple packets. In Snort, the Stream5 preprocessor aggregates multiple packets from a given direction of a TCP flow and builds a single packet by concatenating their payloads, allowing rules to match patterns that span packet boundaries. This is accomplished by keeping a descriptor for each active TCP session and tracking the state of the session according to the semantics of the TCP protocol. Stream5 also keeps copies of the packet data and periodically "flushes" the stream by reassembling all contents and emitting a large pseudo-packet containing the reassembled data.

Consequently, the size of a pseudo-packet that is created by the Stream5 preprocessor may be up to 65,535 bytes in length, which is the maximum IP packet length. However, assigning the maximum IP packet length as the size of

	0 StateTable Ptr	4 Length	6 Payload 1536
thread k	0x001a0b	3487	Payload
thread k+1	0x001a0b	1957	Payload
thread k+2	0x001a0b	427	Payload
thread k+3	0x02dbd2	768	Payload

Fig. 4. Matching packets that exceed the MTU size

each row of the buffer would result in a huge, sparsely populated array. Copying the whole array to the device would result in high communication costs, limiting overall performance.

A different approach for storing reassembled packets that exceed the Maximum Transmission Unit (MTU) size, without altering the dimensions of the array, is to split them down into several smaller ones. The size of each portion of the split packet will be less or equal to the MTU size and thus can be copied in consecutive rows in the array.

Each portion of the split packet is processed by different threads. To avoid missing matches that span multiple packets, whenever a thread searches a split portion of a packet, it continues the search up to the following row (which contains the consecutive bytes of the packet), until a *final* or a *fail* state is reached, as illustrated in Figure 4. While matching a pattern that spans packet boundaries, the state machine will perform regular transitions. However, if the state machine reaches a final or a fail state, then it is obvious that there is no need to process the packet any further, since any consecutive patterns will be matched by the thread that was assigned to search the current portion.

4.2 Compiling PCRE Regular Expressions to DFA State Tables

Many existing tools that use regular expressions have support for converting regular expressions into DFAs [5,1]. The most common approach is to first compile them into NFAs, and then convert them into DFAs. We follow the same approach, and first convert each regular expression into an NFA using the Thompson algorithm [29]. The generated NFA is then converted to an equivalent DFA incrementally, using the *Subset Construction* algorithm. The basic idea of subset construction is to define a DFA in which each state is a set of states of the corresponding NFA. Each state in the DFA represents a set of active states in which the corresponding NFA can be in after some transition. The resulting

DFA achieves $O(1)$ computational cost for each incoming character during the matching phase.

A major concern when converting regular expressions into DFAs is the *state-space explosion* that may occur during compilation [6]. To distinguish among the states, a different DFA state may be required for all possible NFA states. It is obvious that this may cause exponential growth to the total memory required. This is primarily caused by wildcards, e.g. (`.*`), and repetition expressions, e.g. (`a(x,y)`). A theoretical worst case study shows that a single regular expression of length n can be expressed as a DFA of up to $O(\Sigma^n)$ states, where Σ is the size of the alphabet, i.e. 2^8 symbols for the extended ASCII character set [12]. Due to state explosion, it is possible that certain regular expressions may consume large amounts of memory when compiled to DFAs.

To prevent greedy memory consumption caused by some regular expressions, we use a hybrid approach and convert only the regular expressions that do not exceed a certain threshold of states; the remaining regular expressions will be matched on the CPU using NFAs. We track of the total number of states during the incremental conversion from the NFA to the DFA and stop when a certain threshold is reached. As shown in Section 5.2, setting an upper bound of 5000 states per expression, more than 97% of the total regular expressions can be converted to DFAs. The remaining expressions will be processed by the CPU using an NFA schema, just like the default implementation of Snort.

Each constructed DFA is a two-dimensional state table array that is mapped linearly on the memory space of the GPU. The dimensions of the array are equal to the number of states and the size of the alphabet (256 in our case), respectively. Each cell contains the next state to move to, as well as an indication of whether the state is a final state or not. Since transition numbers may be positive integers only, we represent final states as negative numbers. Whenever the state machine reaches into a state that is represented by a negative number, it considers it as a final state and reports a match at the current input offset. The state table array is mapped on the memory space of the GPU, as we describe in the following section.

4.3 Regular Expression Matching

We have investigated storing the DFA state table both as textures in the texture memory space, as well as on the linear global memory of the graphics card. A straightforward way to store the DFA of each regular expression would be to dynamically allocate global device memory every time. However, texture memory can be accessed in a random fashion for reading, in contrast to global memory, in which the access patterns must be coalesced [19]. This feature can be very useful for algorithms like DFA matching, which exhibit irregular access patterns across large datasets. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. As we will see in Section 5.3, the texture memory is 2 to 2.5 times faster than global device memory for input data reads.

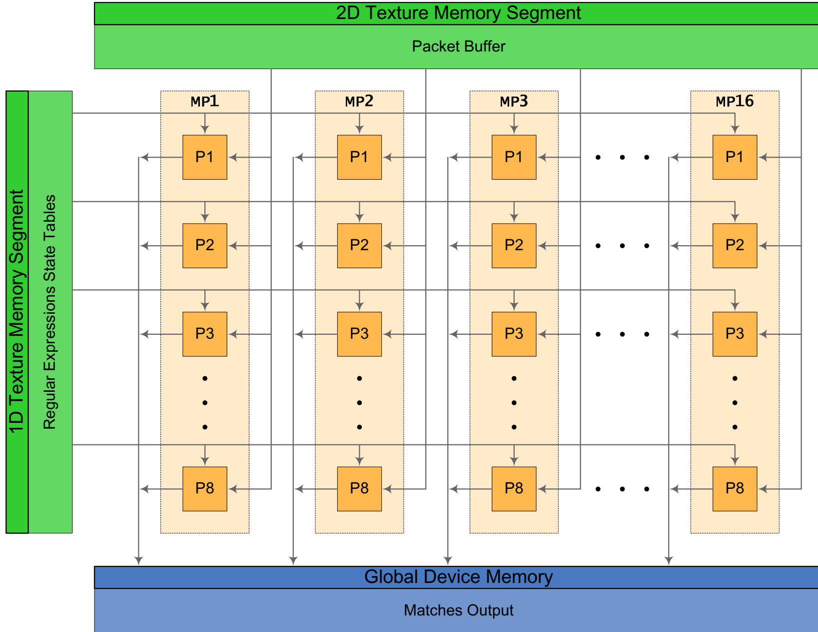


Fig. 5. Regular expression matching on the GeForce 9800 with 128 stream processors. Each processor is assigned a different packet to process using the appropriate DFA.

However, CUDA does not support dynamic binding of memory to texture references. Therefore, it is not feasible to dynamically allocate memory for each state table individually and later bind it to a texture reference. To overcome this limitation, we pre-allocate a large amount of linear memory that is statically bound to a texture reference. All constructed state tables are stored sequentially in this texture memory segment.

During the searching phase, each thread searches a different network packet in isolation, as shown in Figure 5. Whenever a thread matches a regular expression on an incoming packet, it reports it by writing the event to a single-dimension array allocated in the global device memory. The size of the array is equal to the number of packets that are processed by the GPU at once, while each cell of the array contains the position within the packet where the match occurred.

5 Evaluation

5.1 Experimental Environment

For our experiments, we used an NVIDIA GeForce 9800 GX2 card, which consists of two PCBs (Printed Circuit Board), each of which is an underclocked Geforce 8800 GTS 512(G92) video card in SLI Mode. Each PCB contains 128 stream

processors organized in 16 multiprocessors, operating at 1.5GHz with 512 MB of memory. Our base system is equipped with two AMD Opteron^{TM} 246 processors at 2GHz with 1024KB of L2-cache.

For our experiments, we use the following full payload network traces:

U-Web: A trace of real HTTP traffic captured in our University. The trace totals 194MB, 280,088 packets, and 4,711 flows.

SCH-Web: A trace of real HTTP traffic captured at the access link that connects an educational network of high schools with thousands of hosts to the Internet. The trace contains 365,538 packets in 14,585 different flows, resulting to about 164MB of data.

LLI: A trace from the 1998-1999 DARPA intrusion detection evaluation set of MIT Lincoln Lab [2]. The trace is a simulation of a large military network and generated specifically for IDS testing. It contains a collection of ordinary-looking traffic mixed with attacks that were known at the time. The whole trace is about 382MB and consists of 1,753,464 packets and 86,954 flows.

In all experiments, Snort reads the network traces from the local machine. We deliberately chose traces of small size so that they can fit in main memory—after the first access, the whole trace is cached in memory. After that point, no accesses ever go to disk, and we have verified the absence of I/O latencies using the `iostat(1)` tool.

We used the default rule set released with Snort 2.6 for all experiments. The set consists of 7179 rules that contain a total of 11,775 `pcre` regular expressions. All preprocessors were enabled, except the HTTP inspect preprocessor, in order to force all web traffic to be matched against corresponding rules regardless of protocol semantics.

5.2 Memory Requirements

In our first experiment, we measured the memory requirements of our system. Modern graphics cards are equipped with enough and fast memory, ranging from 512MB DDR up to 1.5GB GDDR3 SDRAM. However, the compilation of several regular expression to DFAs may lead to state explosion and consume large amounts of memory.

Figure 6(a) shows the cumulative fraction of the DFA states for the regular expressions of the Snort rule set. It appears that only a few expressions are prone to the state-space explosion effect. By setting an upper bound of 5000 states per regular expression, it is feasible to convert more than 97% of the regular expressions to DFAs, consuming less than 200MB of memory, as shown in Figure 6(b).

5.3 Microbenchmarks

In this section, we analyze the communication overheads and the computational throughput achieved when using the GPU for regular expression matching.

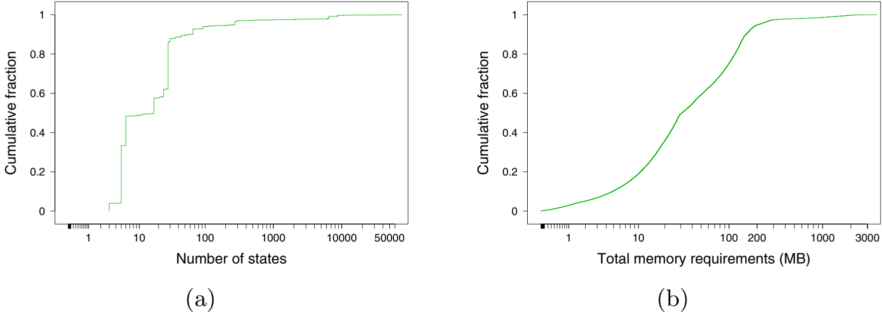


Fig. 6. States (a) and memory requirements (b) for the 11,775 regular expressions contained in the default Snort ruleset when compiled to DFAs

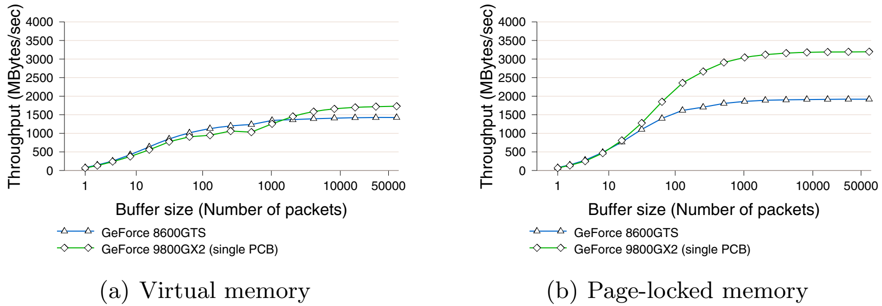
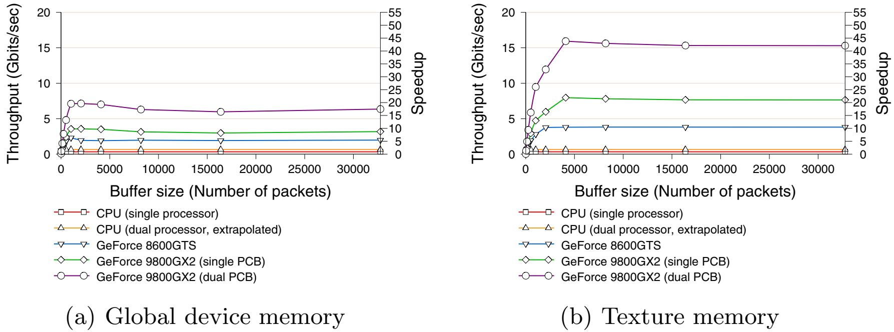


Fig. 7. Sustained throughput for transferring packets to the graphics card using virtual (a) and paged-locked (b) memory

Packet transfer performance. In this experiment we evaluated the time spent in copying the network packets from the memory space of the CPU to the memory space of the GPU. The throughput for transferring packets to the GPU varies depending on the data size and whether page-locked memory is used or not. For this experiment we used two different video cards: a GeForce 8600 operating on PCIe 16x v1.1, and a GeForce 9800 operating on PCIe 16x v2.0.

As expected, copying data from page-locked memory, despite the fact that can be performed asynchronously via DMA, is substantially faster than non page-locked memory, as shown in Figure 7. Compared to the theoretical 4 GB/s peak throughput of the PCIe 16x v1.1 bus, for large buffer sizes we obtain about 2 GB/s with page pinning and 1.5 GB/s without pinning. When using PCIe 16x v2.0, the maximum throughput sustained reached 3.2 GB/s, despite the maximum theoretical being 8 GB/s. We speculate that the reason of these divergences from the theoretical maximum data rates is the use of 8b/10b encoding in the physical layer.



(a) Global device memory (b) Texture memory

Fig. 8. Computational throughput for regular expression matching

Regular expression matching raw throughput. In this experiment, we evaluated the raw processing throughput that our regular expression matching implementation can achieve on the GPU. Thus, the cost for delivering the packets to the memory space of the GPU is not included.

Figure 8 shows the raw computational throughput, measured as the mean size of data processed per second, for both CPU and GPU implementations. We also explore the performance that different types of memory can provide, using both global and texture memory to store the state machine tables. The horizontal axis represents the number of packets that are processed at once by the GPU.

When using global device memory, our GPU implementation operates about 18 times faster than the speed of the CPU implementation for large buffer sizes. The use of texture memory though appears to maximize significantly the utilization of the texture cache. Using texture memory and a 4096 byte packet buffer, the GeForce 9800 achieved an improvement of 48.2 times compared to the CPU implementation, reaching a raw processing throughput of 16 Gbit/s. However, increasing the packet buffer size from 4096 to 32768 packets gave only a slight improvement.

We have also repeated the experiment using the older GeForce 8600GT card which contains only 32 stream processors operating at 1.2GHz. We can see that the achieved performance doubles when going from the previous model to the newest one, which demonstrates that our implementation scales to newer graphics cards.

5.4 Overall Snort Throughput

In our next experiment we evaluated the overall performance of the Snort IDS using our GPU-assisted regular expression matching implementation. Unfortunately, the single-threaded design of Snort forces us to use only one of the two PCBs contained in the GeForce 9800 GX2. Due to the design of the CUDA SDK, multiple host threads are required to execute device code on multiple devices [19]. Thus, Snort's single thread of execution is able to execute device code

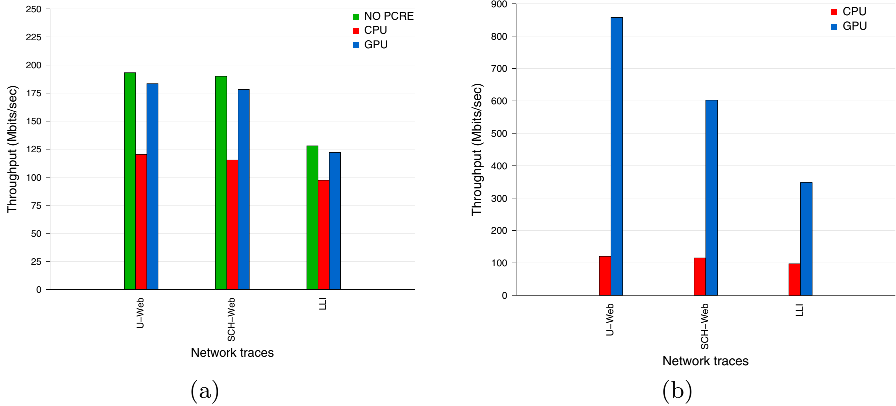


Fig. 9. Sustained processing throughput for Snort using different network traces. In (a) `content` matching is performed on the CPU for both approaches. In (b), both `content` and `pcre` matching is performed on the GPU.

on a single device. It is possible to run multiple instances of Snort dividing the work amongst them, or modify Snort to make it multi-threaded. We are currently in the processes of extending Snort accordingly but this work is beyond the scope of this paper.

We ran Snort using the network traces described in Section 5.1. Figure 9(a) shows the achieved throughput for each network trace, when regular expressions are executed in CPU and GPU, respectively. In both cases, all `content` rules are executed by the CPU. We can see that even when `pcre` matching is disabled, the overall throughput is still limited. This is because `content` rules are executed on the CPU, which limits the overall throughput.

We further offload content rules matching on the GPU using the implementation of GPU string matching from our previous work [30], so that both `content` and `pcre` patterns are matched on the GPU. As we can see in Figure 9(b), the overall throughput exceeds 800 Mbit/s, which is an 8 times speed increase over the default Snort implementation. The performance for the *LLI* trace is still limited, primarily due to the extra overhead spent for reassembling the large amount of different flows that are contained in the trace.

5.5 Worst-Case Performance

In this section, we evaluate the performance of Snort for the worst-case scenario in which each captured packet has to be matched against several regular expressions independently. By sending crafted traffic, an attacker may trigger worst-case backtracking behavior that forces a packet to be matched against more than one regular expressions [25].

We synthetically create worst-case conditions, in which each and every packet has to be matched against a number of regular expressions, by removing all

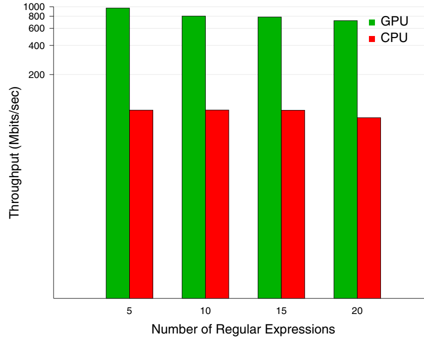


Fig. 10. Sustained throughput for Snort when using only regular expressions

`content` and `uricontent` keywords from all Snort rules. Therefore, Snort's prefiltering pattern matching engine is rendered completely ineffective, forcing all captured packets to be evaluated against each `pcre` pattern individually.

Figure 10 shows how the CPU and the GPU implementations scale as the number of regular expressions increases. We vary the number of `pcre` web rules from 5 to 20, while Snort was operating on the *U-Web* trace. In each run, each packet of the network trace is matched against all regular expressions. Even if the attacker succeeds in causing every packet to be matched against 20 different regular expressions, the overall throughput of Snort remains over 700 Mbit/s when regular expression matching is performed on the GPU. Furthermore, in all cases the sustained throughput of the GPU implementation was 9 to 10 times faster than the throughput on the CPU implementation.

6 Discussion

An alternative approach for regular expression matching, not studied in this paper, is to combine many regular expressions into a single large one. The combination can be performed by concatenating all individual expressions using the logical union operator [28]. However, the compilation of the resulting single expression may exponentially increase the total number of states of the resulting deterministic automaton [16, 26]. The exponential increase, mainly referred as *state-space explosion* in the literature, occurs primarily due to the inability of the DFA to follow multiple partial matches with a single state of execution [15].

To prevent state-space explosion, the set of regular expressions can be partitioned into multiple groups, which can dramatically reduce the required memory space [31, 16]. However, multiple DFAs require the traversal of input data multiple times, which reduces the overall throughput. Recent approaches attempt to reduce the space requirements of the automaton by reducing the number of transitions [16] or using extra scratch memory per state [26, 15]. The resulting automaton is compacted into a structure that consists of a reasonable number of states that are feasible to store in low-memory systems.

Although most of these approaches have succeed in combining all regular expressions contained in current network intrusion detection systems into a small number of automata, it is not straightforward how current intrusion detection systems (like Snort) can adopt these techniques. This is because most of the regular expressions used in attack signatures have been designed such that each one is scanned in isolation for each packet. For example, many expressions in Snort are of the form `/^.{27}/` or `/.{1024}/`, where `.` is the wild card for *any* character followed by the number of repetitions. Such expressions are used for matching the presence of fixed size segments in packets that seem suspicious. Therefore, even one regular expression of the form `/.{N}/` will cause the relevant automaton to generate a huge number of matches in the input stream that need to be checked against in isolation.

Moreover, the combination of regular expressions into a single one prohibits the use of specific modifiers for each regular expression. For example, a regular expression in a Snort rule may use internal information, like the matching position of the previous pattern in the same rule. In contrast, our proposed approach has been implemented directly in the current Snort architecture and boost its overall performance in a straightforward way. In our future work we plan to explore how a single-automaton approach could be implemented on the GPU.

Finally, an important issue in network intrusion detection systems is traffic normalization. However, this is not a problem for our proposed architecture since traffic normalization is performed by the Snort preprocessors. For example, the URI preprocessor normalizes all URL instances in web traffic, so that URLs like "`GET /%43md.exe HTTP/1.1`" become `GET /cmd.exe HTTP/1.1`. Furthermore, traffic normalization can be expressed as a regular expression matching process [22], which can also take advantage of GPU regular expression matching.

7 Related Work

The expressive power of regular expressions enables security researchers and system administrators to improve the effectiveness of attack signatures and at the same time reduce the number of false positives. Popular NIDSes like Snort [21] and Bro [20] take advantage of regular expression matching and come preloaded with hundreds of regexp-based signatures for a wide variety of attacks.

Several researchers have shown interest in reducing the memory use of the compiled regular expressions. Yu et al. [31] propose an efficient algorithm for partitioning a large set of regular expressions into multiple groups, reducing significantly the overall space needed for storing the automata. Becchi et al. [4] propose a hybrid design that addresses the same issue by combining the benefits of DFAs and NFAs. In the same context, recent approaches attempt to reduce the space requirements of an automaton by reducing the number of transitions [16] or using extra scratch memory per state [26, 15].

A significant amount of work focuses on the parallelization of regular expression matching using specialized hardware implementations [9, 24, 7, 18]. Sidhu and Prasanna [24] implemented a regular expression matching architecture for

FPGAs achieving very good space efficiency. Moscola et al. [18] were the first that used DFAs instead of NFAs and demonstrated a significant improvement in throughput.

Besides specialized hardware solutions, commodity multi-core processors have begun gaining popularity, primarily due to their increased computing power and low cost. For highly parallel algorithms, packing multiple cores is far more efficient than increasing the processing performance of a single data stream. For instance, it has been shown that fixed-string pattern matching implementations on SPMD processors, such as the IBM Cell processor, can achieve a computational throughput of up to 2.2 Gbit/s [23].

Similarly, the computational power and the massive parallel processing capabilities of modern graphics cards can be used for non graphics applications. Many attempts have been made to use graphics processors for security applications, including cryptography [11,8], data carving [17], and intrusion detection [14,30,10,27,13]. In our previous work [30], we extended Snort to offload the string matching operations of the Snort IDS to the GPU, offering a three times speedup to the processing throughput compared to a CPU-only implementation. In this work, we build on our previous work to enable both string and regular expression matching to be performed on the GPU.

8 Conclusion

In this paper, we have presented the design, implementation, and evaluation of a regular expression matching engine running on graphics processors, tailored to speed up the performance of network intrusion detection systems. Our prototype implementation was able to achieve a maximum raw processing throughput of 16 Gbit/s, outperforming traditional CPU implementations by a factor of 48. Moreover, we demonstrated the benefits of GPU regular expression matching by implementing it in the popular Snort intrusion detection system, achieving a 60% increase in overall packet processing throughput.

As part of our future work, we plan to run multiple Snort instances in parallel utilizing multiple GPUs instead of a single one. Modern motherboards contain many PCI Express slots that can be equipped with multiple graphics cards. Using a load-balancing algorithm, it may be feasible to distribute different flows to different Snort instances transparently, and allow each instance to execute device code on a different graphics processor. We believe that building such *"clusters"* of GPUs will enable intrusion detection systems to inspect multi-Gigabit network traffic using commodity hardware.

Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS. Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos and Sotiris Ioannidis are also with the University of Crete.

References

1. Pcre: Perl compatible regular expressions, http://www.pcre.org
2. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. ACM Trans. Inf. Syst. Secur. 3(4), 262–294 (2000)
3. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)
4. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: CoNEXT 2007: Proceedings of the 2007 ACM CoNEXT conference, pp. 1–12. ACM, New York (2007)
5. Berk, E., Ananian, C.: Jlex: A lexical analyzer generator for java, http://www.cs.princeton.edu/~appel/modern/java/JLex/
6. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theor. Comput. Sci. 48(1), 117–126 (1986)
7. Clark, C.R., Schimmel, D.E.: Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns, pp. 956–959 (2003)
8. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: Cryptographics: Secret key cryptography using graphics cards. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 334–350. Springer, Heidelberg (2005)
9. Floyd, R.W., Ullman, J.D.: The compilation of regular expressions into integrated circuits. J. ACM 29(3), 603–622 (1982)
10. Goyal, N., Ormont, J., Smith, R., Sankaralingam, K., Estan, C.: Signature matching in network processing using SIMD/GPU architectures. Technical Report TR1628 (2008)
11. Harrison, O., Waldron, J.: Practical symmetric key cryptography on modern graphics hardware. In: Proceedings of the 17th USENIX Security Symposium, Berkeley, CA, USA, July 2008, pp. 195–209. USENIX Association (2008)
12. Hopcroft, J.E., Ullman, J.D.: Introduction To Automata Theory, Languages, And Computation. Addison-Wesley Longman Publishing Co., Inc., Boston (1990)
13. Huang, N.-F., Hung, H.-W., Lai, S.-H., Chu, Y.-M., Tsai, W.-Y.: A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. In: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINA), pp. 62–67
14. Jacob, N., Brodley, C.: Offloading IDS computation to the GPU. In: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference (ACSAC 2006), Washington, DC, USA, pp. 371–380. IEEE Computer Society, Los Alamitos (2006)
15. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: ANCS 2007: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems, pp. 155–164. ACM, New York (2007)
16. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: SIGCOMM 2006: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 339–350. ACM, New York (2006)
17. Richard III, G.G., Marziale, L., Roussev, V.: Massive threading: Using GPUs to increase the performance of digital forensics tools. Digital Investigation 1, 73–81 (2007)

18. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: FCCM, pp. 31–38 (2003)
19. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 1.1, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf
20. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Proceedings of the 7th conference on USENIX Security Symposium (SSYM 1998), Berkeley, CA, USA, p. 3. USENIX Association (1998)
21. Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of the 1999 USENIX LISA Systems Administration Conference (November 1999)
22. Rubin, S., Jha, S., Miller, B.: Protomatching Network Traffic for High Throughput Network Intrusion Detection. In: Proceedings of the 13th ACM conference on Computer and Communications Security (CCS), pp. 47–58
23. Scarpazza, D.P., Villa, O., Petrini, F.: Exact multi-pattern string matching on the cell/b.e. processor. In: CF 2008: Proceedings of the 2008 conference on Computing frontiers, pp. 33–42. ACM, New York (2008)
24. Sidhu, R., Prasanna, V.: Fast regular expression matching using FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2001 (2001)
25. Smith, R., Estan, C., Jha, S.: Backtracking algorithmic complexity attacks against a nids. In: ACSAC 2006: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference, Washington, DC, USA, pp. 89–98. IEEE Computer Society, Los Alamitos (2006)
26. Smith, R., Estan, C., Jha, S.: Xfa: Faster signature matching with extended automata. In: IEEE Symposium on Security and Privacy, pp. 187–201. IEEE Computer Society, Los Alamitos (2008)
27. Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C.: Evaluating GPUs for network packet signature matching. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS (2009)
28. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: CCS 2003: Proceedings of the 10th ACM conference on Computer and communications security, pp. 262–271. ACM, New York (2003)
29. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM 11(6), 419–422 (1968)
30. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High performance network intrusion detection using graphics processors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 116–134. Springer, Heidelberg (2008)
31. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: ANCS 2006: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, pp. 93–102. ACM, New York (2006)

Multi-byte Regular Expression Matching with Speculation

Daniel Luchaup[1], Randy Smith[1], Cristian Estan[2,⋆], and Somesh Jha[1]

[1] University of Wisconsin-Madison
{luchaup,smithr,jha}@cs.wisc.edu
[2] NetLogic Microsystems
estan@netlogicmicro.com

Abstract. Intrusion prevention systems determine whether incoming traffic matches a database of signatures, where each signature in the database represents an attack or a vulnerability. IPSs need to keep up with ever-increasing line speeds, which leads to the use of custom hardware. A major bottleneck that IPSs face is that they scan incoming packets one byte at a time, which limits their throughput and latency. In this paper, we present a method for scanning multiple bytes in parallel using speculation. We break the packet in several chunks, opportunistically scan them in parallel and if the speculation is wrong, correct it later. We present algorithms that apply speculation in single-threaded software running on commodity processors as well as algorithms for parallel hardware. Experimental results show that speculation leads to improvements in latency and throughput in both cases.

Keywords: low latency, parallel pattern matching, regular expressions, speculative pattern matching, multi-byte, multi-byte matching.

1 Introduction

Intrusion Prevention Systems (IPSs) match incoming traffic against a database of signatures, which are Regular Expressions (REs) that capture attacks or vulnerabilities. IPSs are a very important component of the security suite. For instance, most enterprises and organizations deploy an IPS. A significant challenge faced by IPS designers is the need to keep up with ever-increasing line speeds, which has forced IPSs to move to hardware. Most IPSs match incoming packets against signatures one byte at a time, causing a major bottleneck. In this paper we address this bottleneck by investigating the problem of *multi-byte* matching, or the problem of IPS concurrently scanning multiple bytes of a packet. We present a novel speculation-based method for multi-byte matching.

Deterministic Finite Automata (DFAs) are popular for signature matching because multiple signatures can be merged into one large regular expression and a single DFA can be used to match them simultaneously with a guaranteed robust performance of $O(1)$ time per byte. However, matching network traffic against a

⋆ Work done while at University of Wisconsin-Madison.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 284–303, 2009.
© Springer-Verlag Berlin Heidelberg 2009

DFA is inherently a serial activity. We break this inherent serialization imposed by the *pointer chasing* nature of DFA matching using speculation. Speculation has been used in several areas of computer science, especially computer architecture. Our speculative method works by dividing the input into multiple chunks and scanning each of them in parallel using traditional DFA matching. The main idea behind our algorithm is to guess the initial state for all but the first chunk, and then to make sure that this guess does not lead to incorrect results. The insight that makes this work is that although the DFA for IPS signatures can have numerous states, only a small fraction of these states are visited often while parsing benign network traffic. This idea opens the door for an entire new class of parallel multi-byte matching algorithms.

This paper makes the following contributions: We present Speculative Parallel Pattern Matching (SPPM), a novel method for DFA multi-byte matching which can lead to significant speedups. We use a new kind of speculation where gains are obtained not only in the case of correct guesses, but also in the most common case of incorrect ones yet whose consequences quickly turn out to still be valid. Section 3 presents an overview of SPPM, with details given in Section 4. We present a single-threaded SPPM algorithm for commodity processors which improves performance by issuing multiple independent memory accesses in parallel, thus hiding part of the memory latency. Measurements show that by breaking the input into two chunks, this algorithm can achieve an average of 24% improvement over the traditional matching procedure. We present SPPM algorithms suitable for platforms where parallel processing units share a copy of the DFA to be matched. Our models show that when using up to 100 processing units our algorithm achieves significant reductions in latency. Increases in throughput due to using multiple processing units are close to the maximum increase afforded by the hardware.

2 Background

2.1 Regular Expression Matching – A Performance Problem

Signature matching is a performance-critical operation in which attack or vulnerability signatures are expressed as regular expressions and matched with DFAs. For faster processing, DFAs for distinct signatures such as `.*user.*root.*` and `.*vulnerability.*` are combined into a single DFA that simultaneously represents all the signatures. Given a DFA corresponding to a set of signatures, and an input string representing the network traffic, an IPS needs to decide if the DFA accepts the input string. Algorithm 1 gives the procedure for the traditional matching algorithm.

Modern memories have large throughput and large latencies: one memory access takes many cycles to return a result, but one or more requests can be issued every cycle. Suppose that reading $DFA[state][input_char]$ results in a memory access[1] that takes M cycles[2]. Ideally the processor would schedule other

[1] Assuming that the two indexes are combined in a single offset in a linear array.

[2] On average. Caching may reduce the average, but our analysis still holds.

```
   Input: DFA = the transition table
   Input: I = the input string, |I| = length of I
   Output: Does the input match the DFA?
1  state ← start_state;
2  for i = 0 to |I| do
3  |  input_char ← I[i];
4  |  state ← DFA[state][input_char];
5  |  if accepting(state) then
6  |  |  return MatchFound;
7  |  end
8  end
9  return NoMatch;
```

Algorithm 1. Traditional DFA matching

operations while waiting for the result of the read from memory, but in Algorithm 1 each iteration is data-dependent on the previous one: the algorithm cannot proceed with the next iteration before completing the memory access of the current step because it needs the new value for the *state* variable (in compiler terms, M is the Recurrence Minimum Initiation Interval). Thus the performance of the system is limited due to the pointer chasing nature of the algorithm.

If $|I|$ is the number of bytes in the input and if the entire input is scanned, then the duration of the algorithm is at least $M * |I|$ cycles, regardless of how fast the CPU is. This algorithm is purely sequential and can not be parallelized.

Multi-byte matching methods attempt to consume more than one byte at a time, possibly issuing multiple overlapping memory reads in each iteration. An ideal *multi-byte* matching algorithm based on the traditional DFA method and consuming B bytes could approach a running time of $M * |I|/B$ cycles, a factor of B improvement over the traditional algorithm.

2.2 Signature Types

Suffix-closed Regular Expressions over an alphabet Σ are Regular Expressions with the property that if they match a string, then they match that string followed by any suffix. Formally, their language L has the property that $x \in L \Leftrightarrow \forall w \in (\Sigma)^* : xw \in L$. All signatures used by IPSs are suffix-closed. Algorithm 1 uses this fact by checking for accepting states after each input character instead of checking only after the last one. This is not a change we introduced, but a widely accepted practice for IPSs.

Prefix-closed Regular Expressions (PREs) over an alphabet Σ are regular expressions whose language L has the property that $x \in L \Leftrightarrow \forall w \in (\Sigma)^* : wx \in L$. For instance, `.*ok.*stuff.*|.*other.*` is a PRE, but `.*ok.*|bad.*` is not, because the `bad.*` part can only match at the beginning and is not prefix-closed. In the literature, non-PRE signatures such as `bad.*` are also called *anchored* signatures. A large fraction of signatures found in IPSs are prefix-closed.

3 Overview

The core idea behind the *Speculative Parallel Pattern Matching* (SPPM) method is to divide the input into two or more chunks of the same size and process them *in parallel*. We assume that the common case is **not** finding a match, although speedup gains are possible even in the presence of matches. As is customary in IPSs, all our regular expressions are suffix closed. Additionally, at this point we only match REs that are also prefix closed, a restriction that will be lifted in Sec. 4.4. In the rest of this section we informally present the method by example, we give statistical evidence explaining why speculation is often successful, and we discuss ways of measuring and modeling the effects of speculation on latency and throughput.

3.1 Example of Using Speculation

As an example, consider matching the input I=`AVOIDS_VIRULENCE` against the DFA recognizing the regular expression `.*VIRUS` shown in Fig. 1. We break the input into two chunks, I_1=`AVOIDS_V` and I_2=`IRULENCE`, and perform two traditional DFA scans in parallel. A *Primary* process scans I_1 and a *Secondary* process scans I_2. Both use the same DFA, shown in Fig. 1. To simplify the discussion, we assume for now that the Primary and the Secondary are separate processors operating in lockstep. At each step they consume one character from each chunk, for a total of two characters in parallel.

To ensure correctness, the start state of the Secondary should be the final state of the Primary, but that state is initially unknown. We speculate by using the DFA's start state, State 0 in this case, as a start state for the Secondary and rely on a subsequent validation stage to ensure that this speculation does not lead to incorrect results. In preparation for this validation stage the Secondary also records its state after each input character in a *History* buffer.

Figure 2 shows a trace of the two stages of the speculative matching algorithm. During the **parallel processing stage**, each *step i* entry shows for both the Primary and the Secondary the new state after parsing the i-th input character in the corresponding chunk, as well as the history buffer being written by the Secondary. At the end of step 8, the parallel processing stage ends and the Secondary finishes parsing without finding a match. At this point the *History* buffer contains 8 saved states. During the **validation stage**, steps 9-12, the Primary keeps processing the input and compares its current state with the state corresponding to the same input character that was saved by the Secondary in the History buffer. At step 9 the Primary transitions on input 'I' from state 1 to state 2 which is different from 0, the state recorded for that position. Since the Primary and the Secondary disagree on the state after the 9-th, 10-th and 11-th characters, the Primary continues until step 12 when they agree by reaching state 0. Once this *coupling* between the Primary and Secondary happens, it is not necessary for the Primary to continue processing because it would go through the same states and make the same acceptance decisions as the Secondary. We use the term *validation region* to refer to the portion of the input processed by

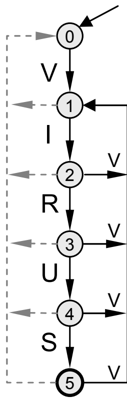


Fig. 1. DFA for `.*VIRUS`; dotted lines show transitions taken when no other transitions apply

Input	A	V	O	I	D	S	_	V	I	R	U	L	E	N	C	E
step 1	**0**								**0**							
step 2		**1**							0	**0**						
step 3			**0**						0	0	**0**					
step 4				**0**					0	0	0	**0**				
step 5					**0**				0	0	0	0	**0**			
step 6						**0**			0	0	0	0	0	**0**		
step 7							**0**		0	0	0	0	0	0	**0**	
step 8								**1**	0	0	0	0	0	0	0	**0**
step 9									$2 \neq 0$,	0	0	0	0	0	0	0
step 10										$3 \neq 0$,	0	0	0	0	0	0
step 11											$4 \neq 0$,	0	0	0	0	0
step 12												$0 = 0$,	0	0	0	0

Fig. 2. Trace for the speculative parallel matching of P=`.*VIRUS` in I=`AVOIDS_VIRULENCE`. During the parallel stage, steps 1-8, the Primary scans the first chunk. The Secondary scans the second chunk and updates the history buffer. The Primary uses the history during validation stage, steps 9-12, while re-scanning part of the input scanned by the Secondary till agreement happens at step 12.

both the Primary and the Secondary (the string `IRUL` in this example). *Coupling* is the event when the validation succeeds in finding a common state.

In our case, the input is 16 bytes long but the speculative algorithm ends after only 12 iterations. Note that for different inputs, such as `SOMETHING_ELSE..`, the speculative method would stop after only 9 steps, since both halves will see only state 0. The performance gain from speculative matching occurs only if the Primary does not need to process the whole input. Although we guess the starting state for the Secondary, performance improvements do not depend on this guess being right, but rather on validation succeeding quickly, *i.e.* having a validation region much smaller than the second chunk.

3.2 Statistical Support for Speculative Matching

In this section we provide an intuitive explanation behind our approach. We define a *default transition* to be a transition on an input character that does not advance towards an accepting state, such as the transitions shown with dotted lines in Fig. 1. If we look at Fig. 1, we see that the automaton for `.*VIRUS.*` will likely spend most of its time in state 0 because of the default transitions leading to state 0. Figure 2 shows that indeed 0 is the most frequent state. In general, it is very likely that there are just a few *hot* states in the DFA, which are the target states for most of the transitions. This is particularly true for PREs because they start with `.*` and this usually corresponds to an initial state with default transitions to itself.

For instance, we constructed the DFA composed from 768 PREs from Snort and measured the state frequencies when scanning a sample of real world HTTP

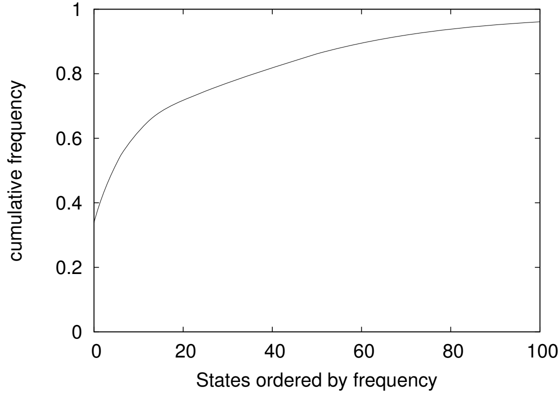


Fig. 3. The state frequency CDF graph for a PRE composed of 768 Snort signatures. Most of the scanning time is spent in a few hot states. The most frequent state accounts for 33.8% of the time and the first 6 most frequent states account for half the time.

traffic. Fig. 3 displays the resulting *Cumulative Distribution Function* (CDF) graph when the states are ordered in decreasing order of frequency. Most time is spend in a relatively small number of states. The most frequent state occurs in 33.8% of all transitions, and the first 6 states account for 50% of the transitions.

The key point is that there is a state that occurs with a relatively high frequency, 33.8% in our case. A back-of-the-envelope calculation shows that it is quite likely that both halves will soon reach that state. Indeed, assume a pure probabilistic model where a state S occurs with a 33.8% probability at any position. The chances for coupling due to state S at a given position are $0.338^2 = 0.114$. Equivalently, the chances that such coupling doesn't happen are $1 - 0.338^2 = 0.886$. However, the chances that disagreement happens on each of h consecutive positions are 0.886^h, which diminishes quickly with h. The probability for coupling in one of 20 different positions is $1 - 0.886^{20} = 0.912$. Even if the frequency of a state S was 5% instead of 33.8%, it would take 45 steps to have a probability greater than 90% for two halves to reach state S. While 45 steps may seem high, it is only a tiny fraction, 3%, compared to the typical maximum TCP packet length of 1500 bytes. In other words, we contend that the length of the validation region will be small.

Note that the high probability of coupling in a small number of steps is based on a heavily biased distribution of frequencies among the N states of the DFA. If all states were equally probable, then the expected number of steps to coupling would be $O(N)$. This would make coupling extremely unlikely for automata with large numbers of states.

3.3 Performance Metrics

One fundamental reason why speculation improves the performance of signature matching is that completing two memory accesses in parallel takes less time

than completing them serially. While the *latencies* of memories remain large, the achievable *throughput* is high because many memory accesses can be completed in parallel.

When we apply SPPM in single-threaded software settings, the processing time for packets determines both the throughput and the latency of the system as packets are processed one at a time. Our measurements show that SPPM improves both latency and throughput. When compared to other approaches using a parallel architecture, SPPM improves latency significantly and achieves a throughput close to the limits imposed by hardware constraints.

4 Speculative Matching

Speculative Parallel Pattern Matching is a general method. Depending on the hardware platform, the desired output, the signature types, or other parameters, one can have a wide variety of algorithms based on SPPM. This section starts by formalizing the example from Section 3.1 and by introducing a simplified performance model for evaluating the benefits of speculation. After presenting basic SPPM algorithms for single-threaded software and for simple parallel hardware, we discuss variants that are not constrained by the simplifying assumptions. These generalized algorithms work with unconstrained regular expressions, return more information about the match, not just whether a match exists or not, and limit speculation to guarantee good worst-case performance.

4.1 Basic SPPM Algorithm

Algorithm 2 shows the pseudocode for the informal example from Sect. 3.1. The algorithm processes the input in three stages.

During the **initialization stage** (lines 1-5), the input is divided into two chunks and the state variables for the Primary and Secondary are initialized. During the **parallel processing stage** (lines 6-13), both processors scan their chunks in lockstep. If either the Primary or the Secondary reach an accepting state (line 10), we declare a match and finish the algorithm (line 11). The Secondary records (line 12) the states it visits in the history buffer (for simplicity, the history buffer is as large as the input, but only its second half is actually used). During the **validation stage** (lines 14-21), the Primary continues processing the Secondary's chunk. It still must check for accepting states as it may see a different sequence of states than the Secondary. There are three possible outcomes: a match is found and the algorithm returns success (line 18), coupling occurs before the end of the second chunk (line 20) or the entire second chunk is traversed again. If the input has an odd number of bytes, the first chunk is one byte longer, and a sentinel is setup at line 5 such that the validation step will ignore it.

Correctness of Algorithm 2. If during the parallel processing stage the Secondary reaches the **return** at line 11, then the Secondary found a match on its chunk. Since our assumption is that we search for a prefix-closed regular expression, a match in the second chunk guarantees a match on the entire input. Therefore it is safe to return with a match.

```
   Input: DFA = the transition table
   Input: I = the input string
   Output: Does the input match the DFA?
   // Initialization stage
 1 len ← |I| ;                                              // Input length
 2 (len_1, len_2) ← (⌈len/2⌉, ⌊len/2⌋);                     // Chunk sizes
 3 (chunk_1, chunk_2) ← (&I, &I + len_1);                   // Chunks
 4 (S_1, S_2) ← (start_state, start_state);                 // Start states
 5 history[len_1 − 1] ← error_state ;                       // Sentinel
   // Parallel processing stage
 6 for i = 0 to len_2 − 1 do
 7     forall k ∈ {1, 2} do in parallel
 8         c_k ← chunk_k[i];
 9         S_k ← DFA[S_k][c_k];
10         if accepting(S_k) then
11             return MatchFound;
12     history[len_1 + i] ← S_2 ;                           // On Secondary
13     i ← i + 1;
   // Validation stage (on Primary)
14 while i < len do
15     c_1 ← I[i];
16     S_1 ← DFA[S_1][c_1];
17     if accepting(S_1) then
18         return MatchFound ;
19     if S_1 == history[i] then
20         break;
21     i ← i + 1;
22 return NoMatch ;                                         // Primary finished processing
```

Algorithm 2. Parallel SPPM with two chunks. Accepts PREs.

If the algorithm executes the **break** at line 20, then the Primary reaches a state also reached by the Secondary. Since the behavior of a DFA depends only on the current state and the rest of the input, we know that if the Primary would continue searching, from that point on it would redundantly follow the steps of the Secondary which did not find a match, so it is safe to break the loop and return without a match.

In all the other cases, the algorithm acts like an instance of Algorithm 1 performed by the Primary where the existence of the Secondary can be ignored.

To conclude, Algorithm 2 reports a match if and only if the input contains one.

Simplified performance models. Our evaluation of SPPM includes actual measurements of performance improvements on single-threaded software platforms. But to understand the performance gains possible through speculation

Table 1. Simplified performance model metrics (N is number of processors)

Metric	Definition
Useful work	Number of bytes scanned, $\|I\|$
Processing latency (L)	Number of parallel steps/iterations
Speedup (S)	$S = \|I\|/L$
Processing cost (P)	$P = N \cdot L$
Processing efficiency (P_e)	$P_e = \|I\|/(N \cdot L)$
Memory cost (M)	Number of accesses to DFA table
Memory efficiency (M_e)	$M_e = \|I\|/M$
Size of validation region (V)	Number of steps in validation stage

and to estimate the performance for parallel platforms with different bottlenecks we use a simplified model of performance. Because the input and the history buffer are small (1.5KB for a maximum-sized packet) and are accessed sequentially they should fit in fast memory (cache) and we do not account for accesses to them. We focus our discussion and our performance model on the accesses to the DFA table. Table 1 summarizes the relevant metrics.

We use the number of steps (iterations) in the parallel processing, $|I|/2$, and in the validation stage, V, to approximate the *processing latency*: $L = \frac{|I|}{2} + V$.

Each of these iterations contains one access to the DFA table. The latency of processing an input I with the traditional matching algorithm (Algorithm 1) would be $|I|$ steps, hence we define the *speedup* (latency reduction) as $S = \frac{|I|}{L} = \frac{|I|}{|I|/2+V} = \frac{2}{1+2V/|I|}$.

The *useful work* performed by the parallel algorithm is scanning the entire input, therefore equivalent to $|I|$ serial steps. This is achieved by using $N = 2$ processing units (PUs), the Primary and Secondary, for a duration of L parallel steps. Thus, the amount of processing resources used (assuming synchronization between PUs), the *processing cost* is $P = N \cdot L$ and we define the *processing efficiency* as $P_e = \frac{useful\ work}{processing\ cost} = \frac{|I|}{N \cdot L} = \frac{|I|}{2\cdot(|I|/2+V)} = \frac{1}{1+2V/|I|}$.

Another potential limiting factor for system performance is memory throughput: the number of memory accesses that can be performed during unit time. We define *memory cost*, M, as the number of accesses to the DFA data structure by all PUs, $M = |I| + V$. Note that $M \leq N \cdot L$ as during the validation stage the Secondary does not perform memory accesses. We define *memory efficiency* as $M_e = \frac{|I|}{M} = \frac{|I|}{|I|+V} = \frac{1}{1+V/|I|}$ and it reflects the ratio between the throughput achievable by running the reference algorithm in parallel on many packets and the throughput we achieve using speculation. Both P_e and M_e can be used to characterize system throughput: P_e is appropriate when tight synchronization between the PUs is enforced (e.g. SIMD architectures) and the processing capacity is the limiting factor, M_e is relevant when memory throughput is the limiting factor.

Performance of Algorithm 2. In the worst case, no match is found, and coupling between Primary and Secondary doesn't happen ($V = |I|/2$). In this

```
    Input: DFA = the transition table
    Input: I = the input string
    Output: Does the input match the DFA?
    // Initialization as in Algorithm 2
 1  ...
 6  for i = 0 to len2 − 1 do
 7  |   c1 ← chunk1[i];
 8  |   c2 ← chunk2[i];
 9  |   S1 ← DFA[S1][c1];
10  |   S2 ← DFA[S2][c2];
11  |   if accepting(S1)||accepting(S2) then
12  |   |_ return MatchFound;
13  |   history[len1 + i] ← S2;
14  |_  i ← i + 1;
    // Validation as in Algorithm 2
15  ...
```

Algorithm 3. Single-threaded SPPM with two chunks. Accepts PREs.

case the Primary follows a traditional search of the input and all the actions of the Secondary are overhead. We get $L = |I|$, $S = 1$, $P_e = 50\%$, $M = 1.5|I|$, and $M_e = 67\%$. In practice, because the work during the iterations is slightly more complex than for the reference algorithm (the secondary updates the history), we can even get a small slowdown, but the latency cannot be much lower than that of the reference algorithm.

In the common case, no match occurs and $V \ll |I|/2$. We have $S = \frac{2}{1+2V/|I|}$, $P_e = \frac{1}{1+2V/|I|}$, $M = |I| + V/|I|$, and $M_e = \frac{1}{1+V/|I|}$, where $V/|I| \ll 1$. Thus the latency is typically close to half the latency of the reference implementation and the throughput achieved is very close to that achievable by just running the reference implementation in parallel on separate packets.

In the uncommon case where matches are found, the latency is the same as for the reference implementation if the match is found by the Primary. If the match is found by the Secondary, the speedup can be much larger than 2.

4.2 SPPM for Single-Threaded Software

Algorithm 3 shows how to apply SPPM for single-threaded software. We simply rewrite the parallel part of Algorithm 2 in a serial fashion with the two table accesses placed one after the other. Except for this serialization, everything else is as in Algorithm 2 and we omit showing the common parts. The duration of one step (lines 6-14) increases and the number of steps decreases as compared to Algorithm 1. The two memory accesses at lines 9-10 can overlap in time, so the duration of a step increases but does not double. If the validation region is small, the number of steps is little over half the original number of steps. The reduction in the number of steps depends only on the input and on the DFA whereas the

increase in the duration of a step also depends on the specific hardware (processor and memory). Our measurements show that speculation leads to an overall reduction in processing time and the magnitude of the reduction depends on the platform. The more instructions the processor can execute during a memory access, the larger the benefit of speculation.

This algorithm can be generalized to work with $N > 2$ chunks, but the number of variables increases (e.g. a separate state variable needs to be kept for each chunk). If the number of variables increases beyond what can fit in the processor's registers, the overall result is a slowdown. We implemented a single-threaded SPPM algorithm with 3 chunks, but since its performance is weaker on the platforms we evaluated, we only report results for the 2-chunk version.

4.3 SPPM for Parallel Hardware

Algorithm 4 generalizes Algorithm 2 for the case where N PUs work in parallel on N chunks of the input. We present this unoptimized version due to its simplicity.

Lines 2-5 initialize the PUs. They all start parsing from the initial state of the DFA. They are assigned starting positions evenly distributed in the input buffer: PU$_k$ starts scanning at position $\lfloor (k-1) * |I|/N \rfloor$. During the **parallel processing stage** (lines 6-13) all PUs perform the traditional DFA processing for their chunks and record the states traversed in history (this is redundant for PU_1). The first $N-1$ PUs participate in the **validation stage** (lines 14-25). A PU stops (becomes inactive) when *coupling* with the right neighbor happens, or when it reaches the end of the input. Active PUs perform all actions performed during normal processing (including updating the history).

The algorithm ends when all PUs become inactive.

Linear History Is Relatively Optimal. Algorithm 4 uses a linear history: for each position in the input, exactly one state is remembered – the state saved by the most recent PU that scanned that position. Thus PU_k sees the states saved by PU_{k+1}, which overwrite the states saved by $PU_{k+2}, ..., PU_{k+3}, ..., PU_N$.

Since we want a PU to stop as soon as possible, a natural question arises: would PU_k have a better chance of *coupling* if it checked the states for *all* of $PU_{k+1}, PU_{k+2}, ..., PU_N$ instead of just PU_{k+1}? Would a 2-dimensional history that saves the set of all the states obtained by preceding PUs at a position offer better information than a linear history that saves only the most recent state? In what follows we show that the answer is *no*: the most recent state is also the most accurate one. If for a certain input position, PU_k agrees with any of $PU_{k+1}, PU_{k+2}, ..., PU_N$ then PU_k must also agree with PU_{k+1} at that position. We obtain this by substituting in the following theorem $chunk_k$ for w_1, the concatenation of chunks $k+1$ to $k+j-1$ for w_2 and any prefix of $chunk_{k+j}$ for w_3. We use the notation $w_1 w_2$ to represent the concatenation of strings w_1 and w_2; and $\delta(S, w)$ to denote the state reached by the DFA starting from state S and transitioning for each character in string w.

Theorem 1 (monotony of PRE parsing). *Assume that DFA is the minimized deterministic finite automaton accepting a prefix-closed regular expression, with*

```
    Input: DFA = the transition table
    Input: I = the input string (|I| =input length)
    Output: Does the input match the DFA?
 1  len ← |I|;
 2  forall PU_k, k ∈ {1..N} do in parallel
 3      index_k ← start position of k-th chunk;
 4      state_k ← start_state;
 5  history[0..len − 1] ← error_state;                                // sentinel
    // Parallel processing stage
 6  while index_1 < ⌊|I|/N⌋ do
 7      forall PU_k, k ∈ {1..N} do in parallel
 8          input_k ← I[index_k];
 9          state_k ← DFA[state_k][input_k];
10          if accepting(state_k) then
11              return MatchFound;
12          history[index_k] = state_k;
13          index_k ← index_k + 1;
14  forall PU_k, k ∈ {1..N − 1} do in parallel active_k ← true ;
15  while there are active PUs do
16      forall PU_k such that (active_k == true) do in parallel
17          input_k ← I[index_k];
18          state_k ← DFA[state_k][input_k];
19          if accepting(state_k) then
20              return MatchFound;
21          if history[index_k] == state_k OR index_k == len − 1 then
22              active_k ← false;
23          else
24              history[index_k] = state_k;
25              index_k ← index_k + 1;
26  return NoMatch;
```

Algorithm 4. SPPM with N processing Units (PUs). Accepts PREs.

S_0 *= the start state of the DFA. For any* w_1, w_2, w_3 *input strings we have:* $\delta(S_0, w_1w_2w_3) = \delta(S_0, w_3) \Rightarrow \delta(S_0, w_1w_2w_3) = \delta(S_0, w_2w_3)$.

Proof. Let $S_1 = \delta(S_0, w_1w_2w_3) = \delta(S_0, w_3)$ and $S_2 = \delta(S_0, w_2w_3)$. Assume, by contradiction, that $S_1 \neq S_2$. Since DFA is minimal, there must be a string w such that only one of $\delta(S_1, w)$ and $\delta(S_2, w)$ is an accepting state and the other one is not.

Assume L = the language accepted by the DFA.

We have two cases:

1. $\delta(S_1, w)$ accepting and $\delta(S_2, w)$ is not. Since $\delta(S_1, w) = \delta(\delta(S_0, w_3), w) = \delta(S_0, w_3w)$ we have $\delta(S_1, w)$ accepting $\Rightarrow \delta(S_0, w_3w)$ accepting. Hence $w_3w \in$

L. Since L is prefix closed, $w_3 w \in L \Rightarrow w_2 w_3 w \in L \Rightarrow \delta(S_0, w_2 w_3 w)$ accepting. But $\delta(S_0, w_2 w_3 w) = \delta(\delta(S_0, w_2 w_3), w) = \delta(S_2, w)$. Therefore $\delta(S_2, w)$ is accepting, which is a contradiction.

2. $\delta(S_2, w)$ is accepting and $\delta(S_1, w)$ is not. Then $\delta(S_2, w) = \delta(\delta(S_0, w_2 w_3), w) = \delta(S_0, w_2 w_3 w)$ is accepting. Hence $w_2 w_3 w \in L$. Since L is prefix closed, $w_2 w_3 w \in L \Rightarrow w_1 w_2 w_3 w \in L$. We have $w_1 w_2 w_3 w \in L \Leftrightarrow \delta(S_0, w_1 w_2 w_3 w)$ is accepting. But, $\delta(S_0, w_1 w_2 w_3 w) = \delta(\delta(S_0, w_1 w_2 w_3), w) = \delta(S_1, w)$. Therefore $\delta(S_1, w)$ is accepting, which is also a contradiction.

Both cases lead to contradiction, so our assumption was wrong and $S_1 = S_2$. □

Performance of Algorithm 4. We define *validation region k* as the portion of the packet processed by PU_k during validation, so it can go beyond the end of chunk $k+1$. Let V_k be the length of the validation region k, $V_{max} = \max_{k=1}^{N} V_k$, and $V_\Sigma = \sum_{k=1}^{N} V_k$.

We get the following performance metrics:

$$\begin{array}{ll}
\text{processing latency} & L = \frac{|I|}{N} + V_{max} \\
\text{speedup} & S = \frac{|I|}{L} = \frac{N}{1+N \cdot V_{max}/|I|} \\
\text{processing cost} & P = N \cdot L \\
\text{processing efficiency} & P_e = \frac{|I|}{P} = \frac{1}{1+N \cdot V_{max}/|I|} \\
\text{memory cost} & M = |I| + V_\Sigma \\
\text{memory efficiency} & M_e = \frac{|I|}{M} = \frac{1}{1+V_\Sigma/|I|}
\end{array} \quad (1)$$

In the worst case (no coupling for any of the chunks) $V_k = |I| - k|I|/N$ (ignoring rounding effects), $V_{max} = |I|(1-1/N)$ and $V_\Sigma = (N-1)|I|/2$ which results in a latency of $L = |I|$ (no speedup, but no slowdown either), a processing efficiency of $P_e = 1/N$, and a memory efficiency of $M_e \approx 2/N$. Note that the processing efficiency and the memory efficiency do not need to be tightly coupled. For example if there is no coupling for the first chunk, but coupling happens fast for the others, the latency is still $L = |I|$ and thus $P_e = 1/N$, but $M_e \approx 50\%$ as most of the input is processed twice. But our experiments show that for N below 100, the validation regions are typically much smaller than the chunks and the speedups we get are on the order of $S \approx N$ and efficiencies are $P_e \approx 100\%$ and $M_e \approx 100\%$.

We note here that SPPM always achieves efficiencies of less than 100% on systems using parallel hardware: within our model, the ideal throughput one can obtain by having the PUs work on multiple packet in parallel is always slightly higher than with SPPM. The benefit of SPPM is that the latency of processing a single packet decreases significantly. This can help reduce the size of buffers needed for packets (or the fraction of the cache used to hold them) and may reduce the overall latency of the IPSs which may be important for traffic with tight service quality requirements. Furthermore systems using SPPM can break the workload into fixed-size chunks as opposed to variable-sized packets which simplifies scheduling in tightly coupled SIMD architectures where the processing cost is determined by the size of the largest packet (or chunk) in the

batch. This can ultimately improve throughput as there is no need of batching together packets of different sizes. Due to the complexity of performance in IPSs with parallel hardware, it depends on the specifics of the system beyond those captured by our model whether SPPM, simple parallelization, or a mix of the two is the best way to achieve good performance.

4.4 Relaxing the Assumptions

Matching non-PRE expressions. The basic SPPM algorithms require prefix-closed expressions only because Secondaries are allowed to safely terminate the algorithm if they reach an accepting state. For non-PRE such as `.*ok|bad`, the matches found by Secondaries (which start processing from the start state of the DFA) may be false matches such as the case when the string `bad` occurs at the beginning of the second chunk, not at the beginning of the input. The version of the algorithm described later in this section avoids the problem.

Returning more information about matched packets. The basic matching algorithm is often extended to return more information than just whether a match occurred or not: the offset within the input where the accepting state has been reached and/or the signature number for that matched (a single DFA typically tracks multiple signatures). Furthermore, multiple matches may exist as the reference algorithm may visit accepting states more than once. For example if the DFA recognizes the two signatures `.*day` and `.*week` with a single DFA and the input is `This week on Monday night!`, we have a match for the second signature at the end of the second word and one for the first signature at the end of the fourth word. It is straightforward to extend Algorithm 4 to deliver information about the match, but if the system requires information about the first match (or about all matches), we need a more elaborate modification.

The most general case is when the system requires an ordered list of all matches and accepts arbitrary regular expressions. We change the way Algorithm 4 handles matches: instead of returning immediately, each Secondary PU keeps a list of all the matches it finds. After validation, the individual lists are combined in an ordered list of all matches, but candidate matches found by PU_k at positions preceding the coupling position with PU_{k-1} are discarded. Note that since the common case in IPSs is that no matches are found, the overhead of the extra bookkeeping required is incurred only for a small fraction of the packets and the overall system performance is not affected.

Limiting inefficiency by bounding the validation cost. In the worst case speculation fails and the whole input is traversed sequentially. There is nothing we can do to guarantee a worst case latency smaller than I and equivalently a processing efficiency of more than $1/N$. But we can ensure that the memory efficiency is larger than $2/N$ which corresponds to the case where all PUs traverse the input to the end. We can limit the size of the history buffer to H positions, and stop the validation stage for all PUs other than the primary when they

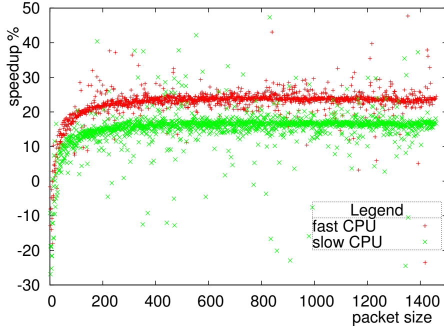


Fig. 4. Speedup of Algorithm 3 over the sequential DFA Algorithm

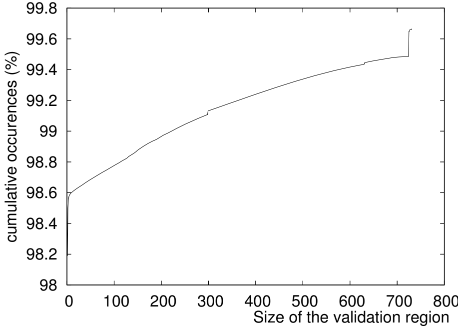


Fig. 5. CDF graph for the sizes of the validation region

reach the end of their history buffer. If H is large enough convergence may still happen (based on our experiments 40 would be a good value), but we bound the number of memory accesses performed during the validation stage to $H(N-2)$ for the $k-2$ non-primary PUs doing validation and $|I| - |I|/N$ for the primary. Thus $M \leq |I|(2-1/N) + H(N-2) < 2|I| + HN$ and $M_e > 1/(2 + HN/|I|)$.

5 Experimental Evaluation

We compared results using speculative matching against the traditional DFA method. We used 106 DFAs recognizing a total of 1499 Snort HTTP signatures. As input we extracted the TCP payloads of 175,668 HTTP packets from a two-hour trace captured at the border router of our department. The most frequent packet sizes were 1448 bytes (50.88%), 1452 bytes (4.62%) and 596 bytes (3.82%). Furthermore 5.73% of the packets were smaller than 250 bytes, 34.37% were between 251 and 1,250 and 59.90% were larger than 1,251.

5.1 Evaluation of Algorithm 3 (Software Implementation)

We implemented Algorithm 3 and measure its actual running time using Pentium performance counters. We ran experiments on two processors, an Intel Core 2 at 2.4GHz and a Pentium M at 1.5GHz. Compared to the traditional sequential algorithms we obtained speedups of 24% and respectively 13%. We explain the higher speedup for the faster processor by the larger gap between the processor speed and the memory latency. Figure 4 shows how the packet size influences the speedup for Algorithm 3: for packets smaller than 20 bytes speculation may result in slowdowns, but for packets larger than 50 bytes the speedup does not change significantly with packet size.

We also find that the validation typically happens quickly. For 98% of the packet validation happens after a single input byte is processed. Validation failed for only 0.335% of the packets. Figure 5 shows the cumulative distribution of the sizes of the validation regions.

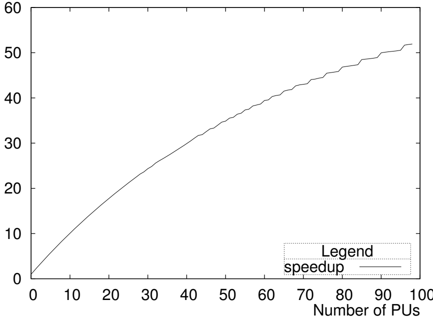


Fig. 6. Speedup for Algorithm 4

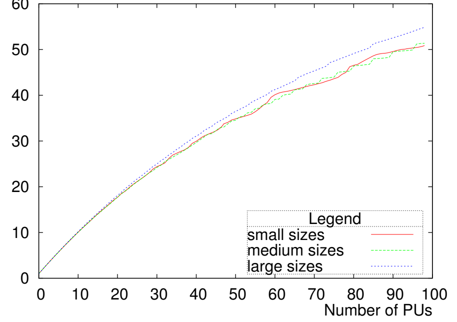


Fig. 7. Speedup by packet size

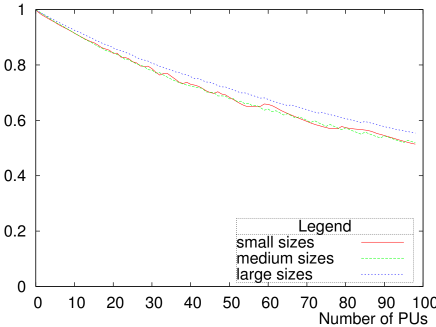


Fig. 8. Processing efficiency by packet size

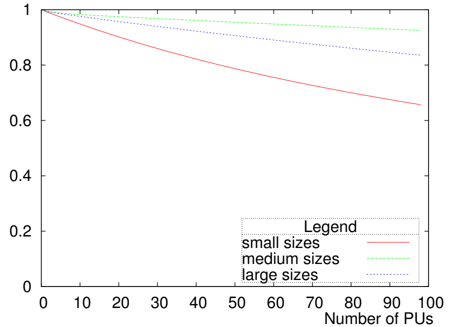


Fig. 9. Memory efficiency by packet size

5.2 Evaluation of Algorithm 4

We evaluated Algorithm 4 for up to $N = 100$ processing units. We report speedups and efficiency based on our performance model which relies on the number of accesses to the DFA data structure (lines 9 and 18 of Algorithm 4). These metrics are described in Sect. 4.3 by equations 1. From Fig. 6 we see that speedup is almost linear up to $N = 20$ and it slowly diverges afterwords. The processing efficiency approaches 50% and the memory efficiency 90% by the time we reach $N = 100$. Figures 7, 8 and 9 show the speedup, processing efficiency and respectively memory efficiency for packets of various sizes: small (1-250), medium (251-1250) and large (1251-1500). The only notable difference is the low memory efficiency for small packets.

Figures 10 and 11 present the cumulative distributions for the sizes of the validation regions when $N = 10$. Figure 10 captures the sizes of all validation regions, which is relevant to memory efficiency. Figure 11 captures only the largest validation region for each packet, which is relevant to processing efficiency. The average size for the validation regions is $V_\Sigma/(N-1) = 2.12$ and for the largest validation regions is $V_{max} = 8.24$. 99.26% of the validation regions were a single byte long and 95.35% of the packet had $V_{max} = 1$.

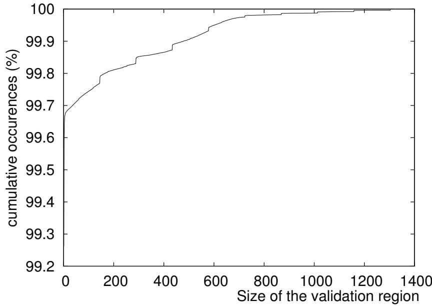


Fig. 10. Cumulative distribution of the sizes of validation regions

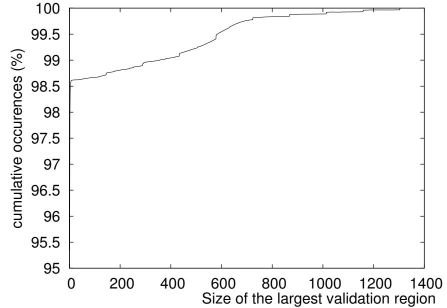


Fig. 11. Cumulative distribution of the size of largest validation regions

6 Related Work

Signature matching is at the heart of intrusion prevention, but traditional matching methods have large memory footprints, slow matching times, or are vulnerable to evasion. Many techniques have been and continue to be proposed to address these weaknesses.

Early string-based signatures used multi-pattern matching algorithms such as Aho-Corasick [1] to efficiently match multiple strings against payloads. Many alternatives and enhancements to this paradigm have since been proposed [27, 8, 25, 16, 26]. With the rise of attack techniques involving evasion [18, 19, 10, 21] and mutation [12], though, string-based signatures have more limited use, and modern systems have moved to vulnerability-based signatures written as regular expressions [28, 6, 24, 20]. In principle, DFA-based regular expression matching yields high matching speeds, but combined DFAs often produce a state-space explosion [22] with infeasible memory requirements. Many techniques have been proposed to reduce the DFA state space [22, 23], or to perform edge compression [15, 3, 13, 9]. These techniques are orthogonal to our own, which focuses specifically on latency and can be readily applied to strings or regular expressions with or without alternative encodings.

Other work uses multi-byte matching to increase matching throughput. Clark and Schimmel [7] and Brodie *et al.* [5] both present designs for multi-byte matching in hardware. Becchi and Crowley [4] also consider multi-byte matching for various numbers of bytes, or *stride*, as they term it. These techniques increase throughput at the expense of changing DFA structure, and some form of edge compression is typically required to keep transition table memory to a reasonable size. Our work on the other hand reduces latency by subdividing a payload and matching the chunks in parallel without changing the underlying automaton. It would be interesting to apply speculative matching to multi-byte structured automata.

Kruegel *et al.* [14] propose a distributed intrusion detection scheme that divides the load across multiple sensors. Traffic is sliced at frame boundaries, and

each slice is analyzed by a subset of the sensors. In contrast, our work subdivides individual packets or flows, speculatively matches each fragment in parallel, and relies on fast validation. Whereas Kruegel's work assumes individual, distinct network sensors, our work can benefit from the increasing availability of multi-core, SIMD, and other n-way processing environments.

Parallel algorithms for regular expression and string matching have been developed and studied outside of the intrusion detection context. Hillis and Steele [11] show that an input of size n can be matched in $\Omega(log(n))$ steps given $n * a$ processors, where a is the alphabet size. Their algorithm handles arbitrary regular expressions but was intended for Connection Machines-style architectures with massive numbers of available processors. Similarly, Misra [17] derives an $O(log(n))$-time string matching algorithm using $O(n*length(string))$ processors. As with the above, the resulting algorithm requires a large number of processors.

Many techniques have been proposed that use Ternary Content addressable Memories (TCAMs). Alicherry *et al.* [2] propose a TCAM-based multi-byte string matching algorithm. Yu *et al.* [30] propose a TCAM-based scheme for matching simple regular expressions or strings. Weinsberg *et al.* [29] introduces the Rotating TCAM (RTCAM), which uses shifted patterns to increase matching speeds further. In all TCAM approaches, pattern lengths are limited to TCAM width and the complexity of acceptable regular expressions is greatly limited. TCAMs do provide fast lookup, but they are expensive, power-hungry, and have restrictive limits on pattern complexity that must be accommodated in software. Our approach is not constrained by the limits of TCAM hardware and can handle regular expressions of arbitrary complexity.

7 Conclusions

We presented speculative pattern matching method which is a powerful technique for low latency regular-expression matching. The method is based on three important observations. The first key insight is that the serial nature of the memory accesses is the main latency-bottleneck for a traditional DFA matching. The second observation is that a speculation that doesn't have to be right from the start can break this serialization. The third insight, which makes such a speculation possible, is that the DFA based scanning for the intrusion detection domain spends most of the time in a few hot states. Therefore guessing the state of the DFA at a certain position and matching from that point on has a very good chance that in a few steps will reach the "correct" state. Such guesses are later on validated using a history of speculated states. The payoff comes from the fact that in practice the validation succeeds in a few steps.

Our technique is the first method we are aware of that performs regular-expression matching in parallel. Our results predict that speculation-based parallel solutions can scale very well. Moreover, as opposed to other methods in the literature, our technique does not impose restrictions on the regular-expressions being matched. We believe that speculation is a very powerful idea and other applications of this technique may benefit in the context of intrusion detection.

Acknowledgements. The authors are thankful to the anonymous reviewers and to Mihai Marchidann for their constructive comments.

References

1. Aho, A.V., Corasick, M.: Efficient string matching: An aid to bibliographic search. Communications of the ACM (June 1975)
2. Alicherry, M., Muthuprasannap, M., Kumar, V.: High speed pattern matching for network IDS/IPS. In: ICNP (November 2006)
3. Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: ANCS 2007 (2007)
4. Becchi, M., Crowley, P.: Efficient regular expression evaluation: Theory to practice. In: Proceedings of the 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). ACM, New York (2008)
5. Brodie, B., Cytron, R.K., Taylor, D.: A scalable architecture for high-throughput regular-expression pattern matching. SIGARCH Comput. Archit. News 34(2), 191–202 (2006)
6. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: IEEE Symposium on Security and Privacy (May 2006)
7. Clark, C.R., Schimmel, D.E.: Scalable pattern matching for high-speed networks. In: IEEE FCCM (April 2004)
8. Dharmapurikar, S., Lockwood, J.W.: Fast and scalable pattern matching for network intrusion detection systems. IEEE Journal on Selected Areas in Comm. 24(10), 1781–1792 (2006)
9. Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., Pietro, A.D.: An improved dfa for fast regular expression matching. SIGCOMM Comput. Commun. Rev. 38(5), 29–40 (2008)
10. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: Usenix Security (August 2001)
11. Hillis, W.D., Guy, J., Steele, L.: Data parallel algorithms. Communications of the ACM 29(12), 1170–1183 (1986)
12. Jordan, M.: Dealing with metamorphism. Virus Bulletin Weekly (2002)
13. Kong, S., Smith, R., Estan, C.: Efficient signature matching with multiple alphabet compression tables. In: Securecomm (September 2008)
14. Kruegel, C., Valeur, F., Vigna, G., Kemmerer, R.: Stateful intrusion detection for high-speed networks. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2002, pp. 285–293 (2002)
15. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: ACM SIGCOMM (September 2006)
16. Liu, R., Huang, N., Chen, C., Kao, C.: A fast string-matching algorithm for network processor-based intrusion detection system. Trans. on Embedded Computing Sys. 3(3), 614–633 (2004)
17. Misra, J.: Derivation of a parallel string matching algorithm. Information Processing Letters 85, 255–260 (2003)
18. Paxson, V.: Defending against network IDS evasion. In: Recent Advances in Intrusion Detection, RAID (1999)

19. Ptacek, T., Newsham, T.: Insertion, evasion and denial of service: Eluding network intrusion detection. Secure Networks, Inc. (January 1998)
20. Roesch, M.: Snort - lightweight intrusion detection for networks. In: 13th Systems Administration Conference. USENIX (1999)
21. Shankar, U., Paxson, V.: Active mapping: Resisting nids evasion without altering traffic. In: IEEE Symp. on Security and Privacy (May 2003)
22. Smith, R., Estan, C., Jha, S.: Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM (August 2008)
23. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: IEEE Symposium on Security and Privacy (May 2008)
24. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: ACM CCS (October 2003)
25. Sourdis, I., Pnevmatikatos, D.: Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In: Int. Conf. on Field Programmable Logic and Applications (September 2003)
26. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: ISCA (June 2005)
27. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: IEEE INFOCOM 2004, pp. 333–340 (2004)
28. Wang, H.J., Guo, C., Simon, D., Zugenmaier, A.: Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In: ACM SIGCOMM (August 2004)
29. Weinsberg, Y., Tzur-David, S., Dolev, D., Anker, T.: High performance string matching algorithm for a network intrusion prevention system. In: High Performance Switching and Routing (2006)
30. Yu, F., Katz, R.H., Lakshman, T.: Gigabit rate packet pattern-matching using tcam. In: ICNP, pp. 174–183 (2004)

Toward Revealing Kernel Malware Behavior in Virtual Execution Environments

Chaoting Xuan[1], John Copeland[1], and Raheem Beyah[1,2]

[1] Georgia Institute of Technology
[2] Georgia State University

Abstract. Using a sandbox for malware analysis has proven effective in helping people quickly understand the behavior of unknown malware. This technique is also complementary to other malware analysis techniques such as static code analysis and debugger-based code analysis. This paper presents *Rkprofiler*, a sandbox-based malware tracking system that dynamically monitors and analyzes the behavior of Windows kernel malware. Kernel malware samples run inside a virtual machine (VM) that is supported and managed by a PC emulator. By building its monitoring component into the PC emulator, Rkprofiler is able to inspect each instruction executed by the kernel malware and therefore possesses a powerful weapon against the malware. Rkprofiler provides several capabilities that other malware tracking systems do not. First, it can detect the execution of malicious kernel code regardless of how the monitored kernel malware is loaded into the kernel and whether it is packed or not. Second, it captures all function calls made by the kernel malware and constructs call graphs from the trace files. Third, a technique called *aggressive memory tagging* (AMT) is proposed to track the dynamic data objects that the kernel malware visit. Last, Rkprofiler records and reports the hardware access events of kernel malware (e.g., MSR register reads and writes). Our evaluation results show that Rkprofiler can quickly expose the security-sensitive activities of kernel malware and thus reduces the effort exerted in conducting tedious manual malware analysis.

Keywords: Dynamic Analysis, Rootkit, Emulator.

1 Introduction

When a attacker breaks into a machine and acquires administrator privileges, kernel malware could be installed to serve various attacking purposes (e.g., process hiding, keystroke logging). The complexity of attackers' activity on machines has significantly increased. Rootkits now cooperate with other malware to accomplish complicated tasks. For example, the rootkit Rustock.B has an encrypted spam component attached to its code image in memory. The initialization routine of this rootkit registers a notification routine to the Windows kernel by calling the kernel function PsCreateProcessNotifyRoutine. This notification

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 304–325, 2009.
© Springer-Verlag Berlin Heidelberg 2009

routine is then invoked each time that a new process is created. When detecting the creation of Windows system process Service.exe, Rustock.B decrypts the spam components and injects two threads into the Service.exe process to execute the spam components [7]. Without understanding the behavior of the Rustock.B rootkit, it would be difficult to determine how the spam threads are injected into the Service.exe process. To fully comprehend malicious activitiy on a compromised machine, it is necessary to catch and dissect key malware that attackers have loaded onto the machine. Thus, analyzing rootkits is an inevitable task for security professionals.

Most of the early rootkits were rudimentary in nature and tended to be single-mission, small and did not employ anti-reverse engineering techniques (e.g., obfuscation). These rootkits could be manually analyzed using disassemblers and debuggers. Since rootkit technology is much more mature today, the situation has changed. Rootkits have more capabilities and their code has become larger and more complex. In addition, attackers apply anti-reverse engineering techniques to rootkits in order to prevent people from determining their behavior. Rustock.C is one such example. The security company, Dr. Web, who claimed to be one of the pioneers that provided defense against Rustock.C, took several weeks to unpack and analyze the rootkit [8]. The botnet using Rustock.C was the third largest spam distributor at that time, sending about 30 million spam messages each day. This example illustrates how the cost incurred by the delay of analyzing kernel malware can be huge. As another example, the conficker worm that has infected millions of machines connected to the Internet was reported by several Internet sources [6] [10] (on April 8th 2009) that a heavily encrypted rootkit, probably a keylogger, was downloaded to the victim machines. At the time of the initial submission of this paper for publication, which was three days later, no one had published the details of the rootkit. It is still unclear how severe the damage (e.g., economic, physical) will be as a result of this un-dissected rootkit. Accordingly, developing new approaches for quickly analyzing rootkits is urgent and also critical to defeating most rootkit-involved attacks.

Several approaches have been proposed to address the rootkits analysis problem to some extent. For examples, HookFinder [30] and HookMap [27] are two rootkit hooking detection systems. The former uses dynamic data tainting to detect the execution of hooked malicious code; and the latter applies backward data slicing to locate all potential memory addresses that can be exploited by rootkits to implant hooks. K-tracer [14] is another rootkit analysis system that uses data slicing and chopping to explore the sensitive kernel data manipulation by rootkits. Unfortunately, these systems cannot meet the goal of comprehensively revealing rootkit behavior in a compromised system. Meeting this goal requires answering two fundamental questions: 1) what kernel functions have been called by rootkits?; and 2) what kernel data objects have been visited by rootkits? In the paper, we present a proof-of-concept system, Rkprofiler, in attempt to address these two questions. Rkprofiler is built based on the PC emulator QEMU [5] and analyzes Windows rootkits. The binary translation of QEMU allows Rkprofiler to sandbox rootkits and inspect each executed

malicious instruction. Further, Rkprofiler develops the memory tagging technique to perform just-in-time symbol resolving for memory addresses visited by rootkits. Combining deep inspection capability with the memory tagging capability, Rkprofiler is able to track all function calls and most kernel object accesses made by rootkits.

The rest of paper is structured as follows. We point out the technical challenges for completely revealing rootkit behavior in Section 2. Section 3 gives the overview of the Rkprofiler system, including its major components and malware analysis process. Section 4 presents the technical details of tracking rootkits. Then, we present several case studies in Section 5 and discuss the limitations of Rkprofiler in Section 6. Section 7 surveys related work and Section 8 gives the conclusion of the paper.

2 Challenges

Modern operating systems (OSs) like Windows and Linux utilize two ring levels (ring 0 and 3) provided by X86 hardware to establish the security boundary between the OS and applications. Kernel instructions and application instructions run at ring level 0 and 3 respectively (also called kernel mode and user mode). The execution of special system instructions (INT, SYSENTER and SYSEXIT) allows the CPU to switch between kernel mode and user mode. This isolation mechanism guarantees that applications can only communicate with the kernel through well-defined interfaces (system calls) that are provided by the OS. Many sandbox-based program analysis systems take advantage of this isolation boundary and monitor the system calls made by malware [1] [3]. While this approach is effective to address user-space malware, it fails to address kernel malware. This is because there is no well-defined boundary between benign kernel code and malicious kernel code. Kernel malware possess the highest privileges and can directly read and write any kernel objects and system resource. Moreover, kernel malware may have no constant "identity" - that is, some kernel malware could be drivers and others could be patches to benign kernel software. So the first challenge is how to create a "virtual" boundary between kernel malware and benign kernel software. Rkprofiler overcomes this challenge by using the timing characteristic of malware analysis. Before loading kernel malware, all kernel code is treated as benign code; after loading kernel malware, newly loaded kernel code is considered malicious. Note this "virtual" boundary only isolates code, but not data. This is because the data created by malicious code can also be accessed by benign code, and Rkprofiler does not monitor the operations of benign kernel code for the purpose of design simplicity and better performance.

When monitoring a VM at the hypervisor layer, only hardware-level activities (e.g., memory reads and writes) are observed. To make these observations useful, it is necessary to translate the hardware-level activities to software-level activities. Here, software-level activities refer to using software terms to describe program activities. For example, "local variable X is modified." This translation requirement is also known as the semantic gap problem [9]. This problem

can be expressed as the following: given a memory address, what is its symbol? Automatically finding the symbols for static kernel objects (global variables and functions) is straightforward, but automatically finding the symbols for dynamic kernel objects (data on stack and heap) is challenging. This challenge is not well addressed by previous work. In this paper, we propose a method called *aggressive memory tagging* (AMT) to overcome this challenge. The basic idea of AMT is to perform the symbol resolution at run time and derive the symbols of dynamic kernel objects from other kernel objects whose symbols have been identified. It should be pointed out that Microsoft does not publish all kernel symbols and we can only gather the kernel symbols that are publically available (Microsoft symbol server, DDK documents and some unofficial Internet sources). So the current implementation of Rkprofiler is not able to resolve many unpublished symbols. Nevertheless, we find that it identifies most sensitive available symbols in our evaluation.

3 System Description

Rkprofiler is composed of four software components: *generator*, *controller*, *monitor* and *reporter*. These software components operate in three phases temporally: *pre-analysis*, *analysis* and *post-analysis*. In the pre-analysis phase, the generator collects symbols of native Windows kernel modules (e.g., ntoskrnl.exe, ndis.sys) from the program database (PDB) files available on the Microsoft symbol server [15] and header files in Microsoft's Driver Development Kit (DDK). Two databases are produced by the generator at the end of this stage: type graph and system map. The type graph database contains the data type definitions of native Windows kernel modules. There are six classes of data types: basic type, enum, structure, function, union, and pointer. The data types in the last four classes are considered as composite data types, indicating that a data type includes at least one sub data type. For example, the sub data types of a structure are data types of its data members. In the type graph database, Rkprofiler assigns a unique type ID to each data type. A data type is represented by its type ID, type name, size, class ID and class specific data (e.g., the number of sub data types and their type IDs). The system map database keeps the names, relative virtual addresses and type IDs of global variables and functions used by native Windows kernel modules. In addition, the names and type ID of parameters and the return value for each function are also stored in system map. The generator is comprised of several executables and Perl scripts.

Executing malware and monitoring its behavior are carried out in the analysis phase. Two components of Rkprofiler, controller and monitor, are involved in this phase. The monitor is built into QEMU. The controller is a standalone shell script that sends commands to the monitor via the Linux signal mechanism. Four commands are defined in their communication messages: `RKP_INIT`, `RKP_RUN`, `RKP_STOP` and `RKP_REPORT`, (which are explained shortly). First, a test VM is started and goes into a clean state in which no malware is installed and executed. Then, the controller sends a `RKP_INIT` command to the monitor. After receiving

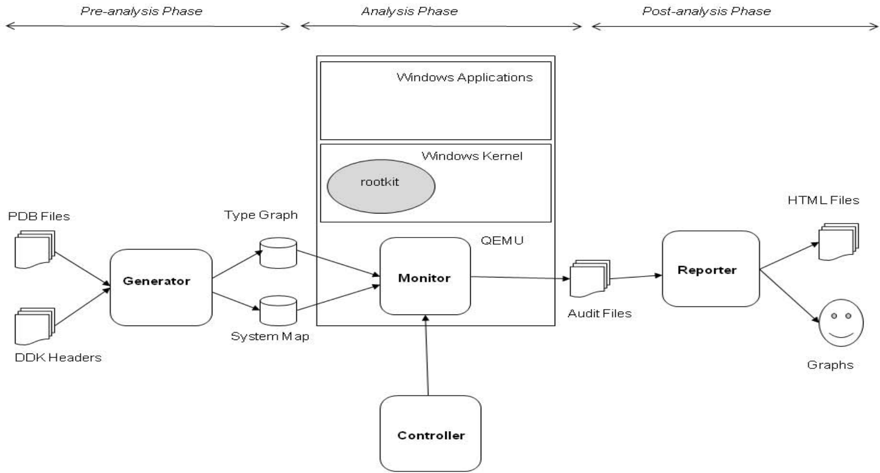


Fig. 1. Rkprofiler architecture and rootkit analysis process

the command, the monitor queries the kernel memory image of the guest OS and creates a hash table of trusted kernel code. Next, the controller instructs the monitor to start monitoring through a `RKP_START` command. At that point, Rkprofiler is ready for the monitoring task. For example, a user starts executing malware in the VM. Depending on the attack objectives of the malware, the user may run other applications to trigger more behaviors from the malware. For example, if the malware is intended to hide processes, the user may open the Windows task manager to induce the hiding behavior. Since the malware can be tested repeatedly, the attack objectives of the malware can be inferred from the analysis results of previous tests. To obtain the monitoring result or end the test, the user can have the controller issue `RKP_REPORT or RKP_STOP` commands to the monitor. The first command informs the monitor to write the monitoring result to local audit files; the second command prompts the monitor to stop monitoring and clear its internal data structures. Four audit files in CSV format are generated in the analysis phase: trace, tag trace, tag access trace, and system resource access trace. These files contain the functions called by the malware, their parameters and return values, kernel data objects visited by the malware and their values. In the post-analysis phase, the reporter is executed to create user-friendly reports. Using the audit files generated in the analysis phase, the reporter performs three tasks. First it builds a call graph from the call trace and saves the graph to another file; second, it visualizes the call graph and tag trace with open-source software GraphViz [11]; third it generates the HTML-formatted reports for call traces and tag traces (CSV format). The entire analysis process is illustrated in Figure 1.

The monitor component of Rkprofiler was built based on the open-source PC emulator QEMU. To support multiple CPU architectures, QEMU defines an

intermediate instruction set. When QEMU is running, each instruction of a VM is translated to the intermediate instructions. Rkprofiler performs code inspection and analysis at the code translation stage. To improve the performance, QEMU caches the translated Translation Block (TB) so that it can be re-executed on the host CPU over time. However, this optimization approach is not desirable to Rkprofiler because an instruction can behave differently in varied machine states. For example, the instruction CALL, whose operand is a general-purpose register, may jump to diverse instructions depending on the value of that register. For each malicious TB that has been cached, Rkprofiler forces QEMU to always perform the code translation. But, the newly generated code is not stored in the cache and the existing cached code is actually executed. Another problem arises when a TB contains multiple instructions. In QEMU, VM states (register and memory contents) are not updated during the TB translation. Except for the first instruction, the translation of all other instructions in a TB could be accompanied by incorrect VM states, possibly resulting in analysis errors. Rkprofiler addresses this problem by making each malicious TB include only one instruction and disabling the direct block chaining for all malicious TBs.

4 Design and Implementation

Kernel malware could take the form of drivers and be legitimately loaded into the kernel. They can also be injected into the kernel by exploiting vulnerabilities of benign kernel software. Rkprofiler is designed to detect kernel malware that enter the kernel in both ways. Roughly speaking, before any malware is executed, Rkprofiler looks up the kernel memory image and identifies all benign kernel code in the VM. Then it groups them into a Trust Code Zone (TCZ) and a hash table is created to store the code addresses of the TCZ. When malware is started, any kernel code that does not belong to the TCZ is regarded as malicious and therefore is tracked by Rkprofiler.

Identification of the trusted kernel code is straightforward if the non-execute (NX) bit of the page table is supported by the (virtual) Memory Management Unit (MMU) of a (virtual) processor. In this case, the kernel code and data do not co-exist in any page of memory. Rkprofiler just needs to traverse the page table of a process to find out all the executable kernel pages. QEMU can provide a NX-bit enabled virtual processor (by enabling the PAE paging mechanism), but this system configuration is not common. Doing so may influence the malware behavior in an undesired manner. For example, the malware could stop running when it detects that the (virtual) CPU is NX enabled. So, the current implementation of Rkprofiler does not require enabling the NX-bit of the virtual CPU. Instead, it interprets all images of benign kernel modules and obtains the Relative Virtual Addresses (RVA) of the code sections. Then it computes their actual virtual addresses by adding the RVAs to the module base addresses, which is acquired by scanning the kernel memory of the VM. After that, Rkprofiler stores the TCZ addresses in a hash table. However, one common type of kernel malware attack is to patch the benign kernel code. To accommodate this

type of attack, Rkprofiler excludes the patched code from the TCZ and revises the TCZ hash table at run time. Rkprofiler identifies the patched code by examining memory write operations and memory copy functions that the malware performs. Note, malware could escape this detection by indirectly modifying the TCZ code (e.g., tampering with kernel memory from user space). A more reliable method is to monitor the integrity of the TCZ as [20] does. Last, Rkprofiler determines whether a kernel TB is malicious or not right before it is translated. If the address of a TB is not within the TCZ, it is deemed as a malicious TB. The hash table implementation of the TCZ ensures that malicious code detection has a small performance hit on the entire system.

4.1 Malicious Code Detection

Kernel malware could take the form of drivers and be legitimately loaded into the kernel. They can also be injected into the kernel by exploiting vulnerabilities of benign kernel software. Rkprofiler is designed to detect kernel malware that enter the kernel in both ways. Roughly speaking, before any malware is executed, Rkprofiler looks up the kernel memory image and identifies all benign kernel code in the VM. Then it groups them into a Trust Code Zone (TCZ) and a hash table is created to store the code addresses of the TCZ. When malware is started, any kernel code that does not belong to the TCZ is regarded as malicious and therefore is tracked by Rkprofiler.

Identification of the trusted kernel code is straightforward if the non-execute (NX) bit of the page table is supported by the (virtual) Memory Management Unit (MMU) of a (virtual) processor. In this case, the kernel code and data do not co-exist in any page of memory. Rkprofiler just needs to traverse the page table of a process to find out all the executable kernel pages. QEMU can provide a NX-bit enabled virtual processor (by enabling the PAE paging mechanism), but this system configuration is not common. Doing so may influence the malware behavior in an undesired manner. For example, the malware could stop running when it detects that the (virtual) CPU is NX enabled. So, the current implementation of Rkprofiler does not require enabling the NX-bit of the virtual CPU. Instead, it interprets all images of benign kernel modules and obtains the Relative Virtual Addresses (RVA) of the code sections. Then it computes their actual virtual addresses by adding the RVAs to the module base addresses, which is acquired by scanning the kernel memory of the VM. After that, Rkprofiler stores the TCZ addresses in a hash table. However, one common type of kernel malware attack is to patch the benign kernel code. To accommodate this type of attack, Rkprofiler excludes the patched code from the TCZ and revises the TCZ hash table at run time. Rkprofiler identifies the patched code by examining memory write operations and memory copy functions that the malware performs. Note, malware could escape this detection by indirectly modifying the TCZ code (e.g., tampering with kernel memory from user space). A more reliable method is to monitor the integrity of the TCZ as [17] does. Last, Rkprofiler determines whether a kernel TB is malicious or not right before it is translated. If the address of a TB is not within the TCZ, it is deemed as a malicious TB. The

hash table implementation of the TCZ ensures that malicious code detection has a small performance hit on the entire system.

4.2 Function Call Tracking

Kernel malware often interacts with the rest of the kernel by calling functions exported by other kernel modules. In Rkprofiler, we use the terms I2E (Internal-to-External) and E2I (External-to-Internal) to describe the function-level control flow transferring between malicious code and benign code. Here, internal and external functions refer to the malicious function code and benign function code respectively. Function calls and returns are two types of events that Rkprofiler monitors. For example, *I2E call* indicates the event that an internal function invokes an external function; *I2E return* refers to the event that an internal function returns to its caller that is an external function. Capturing these function events is important for Rkprofiler to reveal the activity of the malware. Further, in an instance, the kernel malware may directly call the registry functions exported by ntoskrnl.exe like zwSetKeyValue to manipulate local registry entries. Rkprofiler is also designed to capture the I2I (Internal-to-Internal) call and return events. By doing so, Rkprofiler is able to construct (partial) call graphs of the kernel malware, which helps a security professional understand the code structure of the malware. This capability is important, especially when the malware is obfuscated to resist static code analysis. Note, E2E (External-to-External) function events are not monitored here because Rkprofiler does not inspect benign kernel code.

To completely monitor the function-level activity of malware, a data structure called *function descriptor* is defined to represent a stack frame (activation record) of a kernel call stack, allowing Rkprofiler to track the call stacks of the kernel malware. When a function that is called by malware is detected, Rkprofiler creates a new function descriptor object and pushes it to the stack. Conversely, when the function is returned, its function descriptor object is popped from the stack and is deleted. One function descriptor has a pointer that points to the function descriptor of the caller. This pointer is used by Rkprofiler to construct the caller-callee relationships in the post-analysis phase.

The method of detecting a function call event depends on the calling directions. For I2I and I2E calls, Rkprofiler monitors the CALL instructions executed by the malware. Further, it can obtain the function address from the operand of a CALL instruction and the return address that is next to the CALL instruction. For E2I calls, a CALL instruction belongs to TCZ and is not monitored by Rkprofiler. So, the detection point is moved to the first instruction of the callee function. To capture E2I calls, Rkprofiler adds extra data members to the TB descriptor *TranslationBlock*. The first data member indicates what the last instruction of this TB is: CALL, JMP, RET or others. If it is a CALL instruction, the second data member records the return address of the call. Rkprofiler fills in the two data members of a TB when it is being translated. In addition, Rkprofiler creates a global pointer that points to the last TB descriptor whose code was just executed by the virtual CPU. Before translating a malicious TB, Rkprofiler

queries the last TB descriptor to decide if it is an E2I call event. The decision is based on three criteria: 1) if the last TB is benign; 2) if the last instruction of the last TB is CALL; and 3) if the return address stored in the kernel stack is equal to the one stored in the last TB descriptor. The reason for criterion 3 is that the return address is always constant for both direct and indirect calls. On the other hand, Rkprofiler processes the function return events in a similar way to the call events: for I2I and E2I returns, Rkprofiler captures these events by directly monitoring the RET instructions executed by the malware; for I2E returns, Rkprofiler detects them at the instructions directly following the RET instructions and the criteria of the decision are similar to that for the E2I calls.

Two problems complicate the call event detection methods described above. The first one is a *pseudo function call*, which is caused by JMP instructions. When a kernel module attempts to invoke one function exported by another kernel module, it first executes the CALL instruction to invoke an internal stub function and the stub function then jumps to the external function by running the JMP instruction. Normally, the internal stub function is automatically generated by a compiler and the operand of the JMP function is an IAT entry of this module, whose value is determined and inserted by the system loader. Without recognition of these JMP instructions, Rkprofiler incorrectly treats an I2E call as an I2I call: labeling the new function descriptor with the internal stub function address. One example of such functions is DbgPrint. To address a pseudo function call, Rkprofiler first creates an I2I function descriptor and labels it with the internal stub function address. When detecting if an internal JMP instruction is executed in order to jump to an external address, Rkprofiler locates the I2I function descriptor from the top of the function tracking stack, and replaces the internal address with the external address. The second problem is an *interrupt gap*. This is where an interrupt is sent to the (virtual) CPU while it is executing an E2I CALL (or I2E RET) instruction. Consequently, some interrupt handling instructions are executed between the E2I CALL (or I2E RET) instruction and the subsequent internal instruction that Rkprofiler monitors. In this situation, the last TB descriptor does not record the expected CALL (or RET) instruction, so Rkprofiler is unable to track the E2I call (or I2E return) event and observes an unpaired return-call event. The solution to this problem is part of our future work. Fortunately, we did not see interrupt gaps in the experiments.

4.3 Memory Access Tracking

Rkprofiler observes the hardware-level activity of kernel malware, however it should be translated to software-level activity to be understandable to users. Thus, given a virtual address that the malware visits, Rkprofiler is required to find its symbols (e.g., variable name and type). In this paper, we name the process of finding symbols for kernel objects as memory tagging. A memory tag is composed of tag id, virtual address, type ID, variable name (optional) and parent tag id (optional). If a kernel object is owned by the malware, it is an internal kernel object; otherwise, it is an external kernel object. If a kernel object is located in the dynamic memory area (stack and heap), it is a dynamic kernel

object; otherwise, it is a static kernel object. Rkprofiler tags four types of kernel objects: static internal, dynamic internal, static external and dynamic external. Static external kernel objects include global variables and Windows kernel functions. Their symbols are stored in a system map. Tagging a static kernel object is straightforward. Rkprofiler searches the system map by its virtual address and the hit entry contains the target symbols. However, tagging a dynamic kernel object is challenging because its memory is dynamically allocated at run time and the memory address cannot be predicted. Attackers often strip off the symbols of their malware in order to delay reverse engineering, so Rkprofiler assumes that malware samples do not contain valid symbols.

Previous Linux rootkit detection systems [19] [4] present one approach of tracking dynamic kernel objects. A rootkit detector first generates a kernel type graph and identifies a group of global kernel variables. At run time, it periodically retrieves the dynamic objects from the global variables based on the graph type. For example, if a global variable is a linked list head, the detector traverses the list under the direction of the data structure type of list elements. Unfortunately, this approach cannot be applied to the task of profiling kernel malware. First, it covers a limited number of kernel objects, and many other kernel objects such as functions and local variables are not included. Second, since the creation and deletion of dynamic kernel objects could occur at any time, the time gap between every two searches in this approach will produce inaccurate monitoring results. Last, this approach may track many kernel objects that the malware never visits. In this paper, we propose a new symbol exploration approach, *Aggressive Memory Tagging* (AMT), that can precisely find symbols for all kinds of static and dynamic kernel objects at a low computation cost.

AMT Description. We define a kernel object as *contagious* if another kernel object can be derived from it. *Tag inferring* is a process where a kernel object (child object) is derived from another (parent object). Two types of kernel objects are considered contagious: pointers and functions. A pointer kernel object could be a pointer variable or a structure variable containing a pointer member. The child object of a pointer is the pointee object. For a function, its child objects are the parameters and return value of this function. AMT follows the principle of the object tracking approach described above: tracing the dynamic objects from the static objects. Specifically, Rkprofiler first tags all static kernel objects that the malware accesses (memory reads/writes and function calls) by querying the system map. Then, the child objects of the existing contagious tags are tagged via tag inferring. This process is repeated until the malware stops execution or the user terminates monitoring. Note, a tag could become invalid in two scenarios: 1) if when a function returns, the tags of its local variables are invalidated; and 2) if a memory buffer is released, the associated tag becomes out of date as well. Only valid tags can generate valid child tags.

Rkprofiler performs tag inferring through a pointer object at the time that the malware reads or writes the pointer object. The reason is as follows: when reading a pointer, the malware is likely to visit the pointee object through the pointer; when writing a pointer, the malware will possibly modify the pointer to

point to another object if the new value is a valid memory address. Because the executions of benign kernel code are not monitored by Rkprofiler, both read and write operations over a pointer have to be tracked here. If only read operations are monitored, Rkprofiler cannot identify the kernel objects whose pointers are written by malicious code and read by benign code. Many hooks implanted by rootkits fall into this scenario. Similarly, if only write operations are monitored, Rkprofiler can miss the reorganization of kernel objects whose pointers are written by benign code and read by malicious code. Many external kernel objects that are visited by rootkits fall into this scenario. The procedure of tag inferring through a pointer object is as follows: 1) Rkprofiler detects a memory read or write operation and searches the tag queue to check if the target memory corresponds to a contagious tag; 2) if yes, Rkprofiler obtains the up-to-date pointer value and verifies that it is a valid memory address; 3) Rkprofiler searches the tag queue to check if the pointee object is tagged; 4) if not, Rkprofiler obtains the symbols of the pointee object from the type graph and creates a new tag. On the other hand, when a recognizable function is called, tag inferring through the function object is carried out by identifying the function parameters. Input parameters are tagged when the function is called; output parameters are tagged when the function returns.

Implementation. Rkprofiler creates a data structure called *tag descriptor* to represent memory tags. A tag descriptor includes the virtual address of the tag, type ID, a boolean variable, a num variable for memory type, one pointer to the parent tag and one pointer to the function descriptor. The Boolean variable indicates if a tag is contagious or not. The memory type member tells if the tagged object is on the stack, heap or another memory object. Rkprofiler monitors the kernel memory management functions called by malware and records it to a heap list (the memory buffers allocated to the malware). When a buffer is released, Rkprofiler removes it from the heap list. The function descriptor member of a tag helps identify which function is running when this tag is generated. Finally, Rkprofiler maintains a tag queue that contains all the tags that have been created. When a tag is created, its tag descriptor is inserted into the tag queue. The tag is removed from the tag queue after it becomes invalid. Because malware's memory accesses are frequent events, Rkprofiler needs to search the tag queue frequently as well. The tag queue describes a group of various-sized memory segments. If it is organized as a list structure like a linked list, its linear searching time is expensive. To address the problem, Rkprofiler applies the approach presented in [29] that converts a group of various-sized memory segments to a hash table. The basic idea is to break a memory segment into a number of fix-sized memory segments (buckets). A list structure is stored in one bucket to handle the case that some portions of the bucket should not be counted. In this way, the time for searching the tag queue becomes constant.

The Windows kernel provides built-in supports for linked lists via two data structures: `SINGLE_LIST_ENTRY` (for single linked list) and `LIST_ENTRY` (for double linked list). Several kernel APIs are available to simplify driver developers' tasks when managing linked lists (e.g., adding or removing elements). However,

this support causes problems to the memory tagging process of Rkprofiler. For example, in a double linked list, each element contains a data member whose data type is `LIST_ENTRY`. Two pointers of this data member point to the `LIST_ENTRY` data members of two neighbor elements. When one list element is tagged and malware tries to visit the next list element from this one, Rkprofiler just tags the `LIST_ENTRY` data member of the next list element with the type `LIST_ENTRY`. This is not acceptable because what Rkprofiler wants to tag is the next list element with its type. In the pre-analysis stage, we annotated the `SINGLE_LIST_ENTRY` and `LIST_ENTRY` data members with the type names of list elements and their offsets. When parsing the type header file, the generator replaces the `SINGLE_LIST_ENTRY` and `LIST_ENTRY` data members with pointers to list elements. The offset values are also stored in the type graph, allowing the monitor to find the actual addresses of neighbor elements. Another problem is relative pointers. The Windows kernel sometimes uses relative pointers to traverse a list in the following way: the address of the next element is computed by adding the relative pointer and the address of the current element. One example is the data buffer that contains the disk file query result by kernel function NtQueryDirectoryFile. Because these relative pointers are defined as unsigned integer, we also need to label the relative pointers in the kernel type header file such that Rkprofiler can recognize them and properly compute the element addresses.

Rkprofiler has to handle two ambiguous data types that the Windows kernel source uses. The first one is *union*. Union is a data type that contains only one of several alternative members at any given time, and the memory storage required for a union is decided by its largest data member. Unfortunately, guessing which data member of a union should be used at a given time depends on code context, which is hard to automate in Rkprofiler. The second one is generic pointer *pvoid*. Pvoid can be caste to another data type by developers. The actual data type that pvoid points to at a given time is context dependent too. Automatically predicting the pointee data type for pvoid is another challenge. The current default solution is to replace a union with one of its largest members and leave pvoid alone. While performing the analysis, a user can modify the kernel data type header file and change the definition of union or pvoid in terms of his understanding of their running contexts. An automated solution to this problem is part of our future work.

4.4 Hardware Access Monitoring

In comparison to user-space malware, kernel malware is able to bypass the mediation of the OS and directly access low-level hardware resources. In X86 architectures, in addition to the memory and general-purpose registers that kernel malware access through instructions like MOV and LEA, other types of system storage resources could also be visited and manipulated by kernel malware. CPU caches (e.g., TLB) dedicate registers and buffers of I/O controllers. Attackers have developed techniques that take advantage of these hardware resources to devise new attacks. For example, upon a system service (system call) invocation made by a user-space process, Windows XP uses instruction SYSENTER

(for Intel processor) to perform the fast transition from user space to kernel space. The entry point of kernel code (a stub function) is stored in a dedicated register called `IA32_SYSENTER_EIP`, which is one of Model-Specific Registers (MSRs). When executing SYSENTER, the CPU sets the EIP register with the value of `IA32_SYSENTER_EIP`. Then, the kernel stub function is called and it transfers the control to the target system service. To compromise Windows system services, a rootkit could alter the system control-flow path by resetting the `IA32_SYSENTER_EIP` to the starting address of a malicious stub function, and this function can invoke a malicious system service. So, capturing the malware's accesses to these sensitive hardware resources could be essential to comprehend its attacking behavior. Currently, Rkprofiler monitors twenty system instructions that malware might execute. They are not meant to be complete at this point and can be expanded in the future if necessary.

5 Case Studies

5.1 FUTo

FUTo is an enhanced version of the Windows kernel rootkit FU, which uses the technique called Direct Kernel Object Manipulation (DKOM) to hide processes and drivers and change the process privileges. DKOM allows rootkits to directly manipulate kernel objects, avoiding the use of kernel hooks to intercept events that access these kernel objects. For example, a rootkit can delete an item from the `MODULE_ENTRY` list to hide a device driver without affecting the execution of the system. This technique has been applied to many rootkit attacks, such as hiding processes, drivers and communication ports, elevating privilege levels of threads or processes and skewing forensics [12]. In this experiment, FUTo was downloaded from [21] and it included one driver (msdirectx.sys) and one executable (fu.exe). The fu.exe was a command-line application that installed the driver and sent commands to the driver according to the user's instructions. During the test, we executed the fu.exe to accomplish the following tasks: querying the command options, hiding the driver (msdirect.sys) and hiding the process (cmd.exe). After that, we used Windows native system utilities (task manager and driverquery) to verify that the target driver and process did not show up in their reports. The test took less than 3 minutes.

We compared the call graph created by Rkprofiler with the call graph created by IDA-Pro (which uses the static code analysis technique). It was found that the former was the sub-graph of the latter, which is as expected. The tag trace graph of this test is shown in Table 1. The driver msdirectx was executed in four process contexts in the graph. Process 4 (System) is the Windows native process that was responsible for loading the driver misdirectx. The driver initialization routine (with `tag_id` 0) was executed in this process context. The other three processes were associated with FUTo.exe and they communicated with the misdirectx driver to perform the tasks of hiding the driver and process. One important observation is that the major attacking activities have been recorded by Rkprofiler and can be easily identified in the tag trace table by users. To

Table 1. FUTO tag trace table

Tag ID	Address	Type	Parent	Category	Size(bytes)	Process ID	Process Name
0	0xf6b7e7e6	FUNCT_0049_0953_DriverInit	n/a	function	n/a	4	System
1	0x825c3978	DRIVER_OBJECT	n/a	struct	168	4	System
2	0x827cba00	EPROCESS	n/a	struct	608	4	System
3	0x825991c8	EPROCESS	2	struct	608	4	System
4	0x825ce020	EPROCESS	3	struct	608	4	System
5	0xf7b0ec58	PVOID	n/a	pointer	4	4	System
6	0xf6b8b92	PDEVICE_OBJECT	n/a	pointer	4	4	System
7	0xf6b7e722	FUNCT_0049_095B_MajorFunction	1	function	n/a	4	System
8	0xf6b7d43a	FUNCT_00BC_0957_DriverUnload	1	function	n/a	4	System
9	0x8264600	MODULE_ENTRY	1	struct	52	4	System
10	0x82609f18	DEVICE_OBJECT	n/a	struct	184	1920	fu.exe
11	0x8266fc28	IRP	n/a	struct	112	1920	fu.exe
12	0x8266fc03	IRP	n/a	struct	112	1920	fu.exe
13	0x826bc118	IRP	n/a	struct	112	1952	fu.exe
14	0x826bc103	IRP	n/a	struct	112	1952	fu.exe
15	0x826d8288	MODULE_ENTRY	9	struct	52	1952	fu.exe
16	0x8055ab20	MODULE_ENTRY	9	struct	52	1952	fu.exe
17	0x826bc210	IRP	n/a	struct	112	1952	fu.exe
18	0x825d1020	EPROCESS	4	struct	608	1880	fu.exe
19	0x8273a7c8	EPROCESS	18	struct	608	1880	fu.exe
20	0x826eb408	EPROCESS	19	struct	608	1880	fu.exe
21	0x825d5a80	EPROCESS	20	struct	608	1880	fu.exe
22	x825e4da0	EPROCESS	21	struct	608	1880	fu.exe
23	0x825a9668	EPROCESS	22	struct	608	1880	fu.exe
24	0x82695180	EPROCESS	23	struct	608	1880	fu.exe
25	0x825a0da0	EPROCESS	24	struct	608	1880	fu.exe
26	0x82722980	EPROCESS	25	struct	608	1880	fu.exe
27	0x825c27e0	EPROCESS	26	struct	608	1880	fu.exe
28	0x82624bb8	EPROCESS	27	struct	608	1880	fu.exe
29	0x825de980	EPROCESS	28	struct	608	1880	fu.exe
30	0x8248bda0	EPROCESS	29	struct	608	1880	fu.exe
31	0x8264a928	EPROCESS	30	struct	608	1880	fu.exe
32	0x8263a5a8	EPROCESS	31	struct	608	1880	fu.exe
33	0x825d9020	EPROCESS	32	struct	608	1880	fu.exe
34	0xe13ed7b0	HANDLE_TABLE	4	struct	68	1880	fu.exe
35	0x82607d48	ETHREAD	32	struct	600	1880	fu.exe
36	0xe15ca640	HANDLE_TABLE	32	struct	68	1880	fu.exe
37	0xe10a8a08	HANDLE_TABLE	36	struct	68	1880	fu.exe
38	0xe1747cd0	HANDLE_TABLE	36	struct	68	1880	fu.exe

hide itself, the driver msdirectx first reads the address of its module descriptor (with `tag_id` 9) from its driver object (with `tag_id` 1). Then it removes this module descriptor from the kernel `MODULE_ENTRY` list by modifying the Flink and Blink pointers in two neighbor module descriptors (`tag_id` 15 and 16). Similarly,

to conceal process cmd.exe, msdirectx first obtains the process descriptor (with `tag_id` 2) of the current process by calling kernel function IoGetCurrentProcess. Starting from this process descriptor, msdirectx traverses the kernel EPROCESS list to find the process descriptor (with `tag_id` 4) of process csrss.exe. These two steps take place in the System process context. After receiving the command for hiding the cmd.exe process sent by one of the fu.exe processes, msdiretx searches the kernel EPROCESS list, beginning with the process descriptor of csrss.exe. When the process descriptor (with `tag_id` 32) of cmd.exe is found, msdirectx removes it from the kernel EPROCESS list by altering Flink and Blink pointers in two neighbor process descriptors (with `tag_id` 31 and 33). Furthermore, Flink and Blink pointers in the process descriptor of cmd.exe are also modified to prevent the random Blue Screen of Death (BSOD) when exiting the hidden process. To evade the detection of rootkit detectors, FUTo deletes the hidden process from the other three kernel structures: kernel handle table list, handle table of the process csrss.exe and PspCidTable. The first one is a linked list, and the DKOM behavior of FUTo over this kernel structure was captured and displayed in the tag trace graph too (see `tag_id` 36, 37 and 38). The last two kernel structures are implemented as three-dimensional arrays, which is not supported by the current version of Rkprofiler. So, the tag trace graph does not include the modification of these two kernel structures.

Combining Rkprofiler's output with other reports, we discovered other interesting behavior of FUTo. First, FUTo employed an IOCTL mechanism to pass control commands from user space to kernel space. During the driver initialization, a device `\\Device\\msdirectx` was created by calling the kernel function IoCreateDevice. Then a dispatch function (data type `FUNCT_0049_095B_Majorfunction` and `tag_id` 7) was registered to the driver object (with `tag_id` 1) that was assigned to msdirectx by the Windows kernel. This dispatch function was invoked by the kernel I/O manager to process I/O requests issued by the fu.exe processes. By checking the parameters of this dispatch function, we found that the I/O control codes for process and driver concealment tasks are 0x2a7b2008 and 0x2a7b2020. Second, the kernel string function strncmp was called 373 times by one msdirectx function, implying a brute-force searching operation. The first parameter of this function was constant string "System" and the second parameter was 6 bytes of data within the process descriptor of the process System (with `tag_id` 2). Beginning with the address of the process descriptor, the address of the second parameter was increased by one byte each time this string function was called. The purpose of the search was to find the offset of the process name in the EPROCESS structure. This was confirmed by manually checking the FUTo source. It seems that the definition of EPROCESS structure has changed over the Windows versions and the brute-force searching allows FUTo to work with different Windows versions.

5.2 TCPIRPHOOK

Inserting hooks into the kernel to tamper with the kernel control-flow path is one major technique that attackers apply to rootkit attacks. A hooked function can

intercept and manipulate kernel data to serve its malicious aims. TCPIRPHOOK is one such rootkit and it intends to hide the TCP connections from local users. Specifically, this rookit exploits the dispatch function table of the TCP/IP driver object (associated with driver TCPIP.sys) and substitutes a dispatch function with its hook. The hooked function registers another hook to the I/O request packets (IRP) such that the second hook can intercept and modify the query results for network connections. We downloaded the rootkit package from [21] which also included one driver file, irphook.sys. The rootkit was implemented to conceal all http connections (with destination port 80). Before installing the rootkit, we opened Internet Explorer to visit a few websites, and then ran the netstat utility to display the corresponding http connections. We loaded the irphook.sys to the kernel and used netstat to verify that all https connections were gone. In the end, we unloaded the irphook.sys. The test took less than 3 minutes.

The call graph of TCPIRPHOOK is shown in Figure 2. Function 0xf7ab8132 (irphook.sys) was the first hook that was inserted into the 14th entry (IRP_MJ_DEVICE_CONTROL,) of the dispatch function table in the driver TCPIP.sys. The replaced dispatch function was TCPDispatch (address 0xf726fddf) owned by driver TCPIP.sys. The first hook invoked TCPDispatch 15 times in the call graph. In fact, it is common for rootkits to call the original function in a hook, which reduces the coding complexity of the hook. Function 0xfa7b8000 (irphook.sys) was the second hook that was responsible for modifying the query results for network connections. Although the second hook seems to be called by TCPDispatch in the call graph, the actual direct caller of the second hook was IopfCompleteRequest (ntoskrnl.exe). This is because Rkprofiler did not track the benign kernel code and had no knowledge of their call stacks. On the other hand, even the indirect caller-callee relation between TCPDisptch and the second hook can imply that the network connection query caused synchronous IRP processing and completion in the kernel, which is comparable to

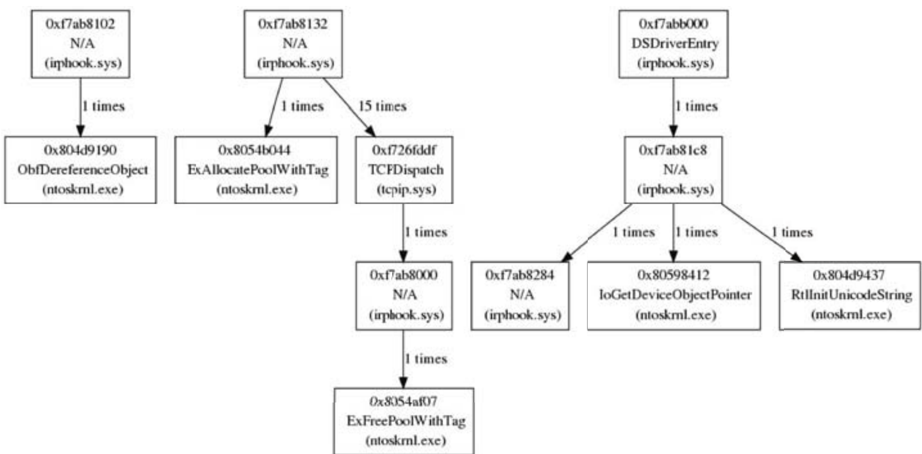


Fig. 2. TCPIRPHOOK call graph

asynchronous IRP processing and completion. But this information cannot be inferred by simply looking at the IDA-pro's call graph, because IDA-pro cannot statically determine the symbol of function TCPDispatch and the calling path from the first hook to the second hook in Figure 3 is not presented in the IDA-pro's call graph.

Figure 3 is the tag trace graph of TCPIRPHOOK. Two hooking activities are illustrated in this graph. The first hook was installed at the driver loading stage. To hook the dispatch function table of the driver TCPIP.sys, TCPIRPHOOK first calls the kernel function IoGetDeviceObjectPointer with the device name `\\Device\\Tcp` to get the pointer (with `tag_id` 7) to the device object (with `tag_id` 8) owned by driver TCPIP.sys. Then, the device object was visited to get the address of the driver object (with `tag_id` 9) owned by driver TCPIP.sys. Last, TCPIRPHOOK carried out the hooking by accessing the 14th entry of the dispatch function table in the driver object: reading the address of the original dispatch function (with `tag_id` 10) and storing it to a global variable; writing the address of the second hook (with `tag_id` 11) to the table entry. The second hook was dynamically installed in the context of process netstat.exe. When netstat.exe was executed to query TCP connection status, the Windows kernel I/O manager created an IRP (with `tag_id` 12) for the netstat.exe process. This IRP was passed to the first hook (`function_id` 5 and `tag_id` 11) of TCPIRPHOOK. The first hook obtained the `IO_STACK_LOCATION` object (with `tag_id` 13) from this IRP and wrote the address of the second hook (with `tag_id` 14) to the data member CompletionRoutine of the `IO_STACK_LOCATION` object. Thus, being one

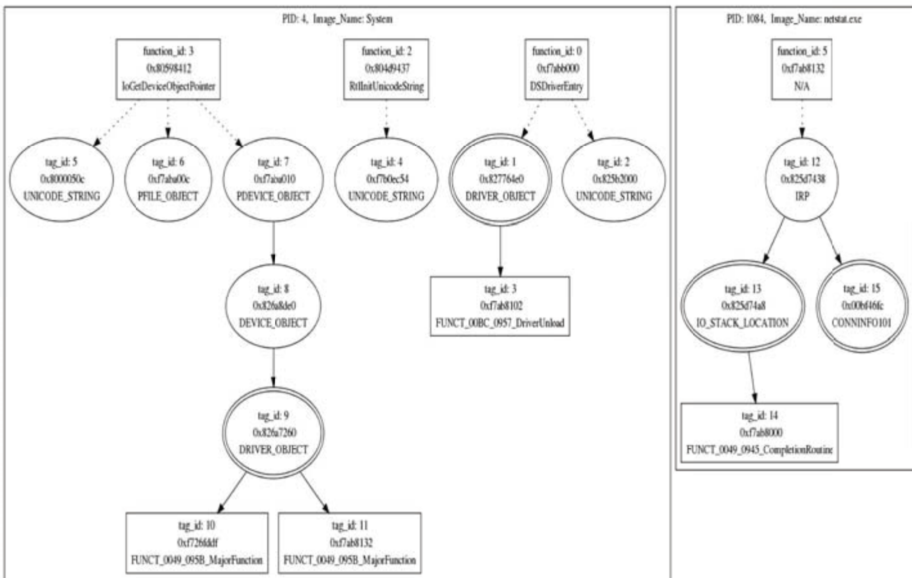


Fig. 3. TCPIRPHOOK tag trace graph

IRP completion function, the second hook would be called by the Windows kernel to process the I/O return data for this IRP. Last, the tag trace graph also captures the manipulation of the I/O return data. The buffer of the I/O return data was pointed to by the data member UserBuffer of IRP and it was an array of structure CONNINF101 (with `tag_id` 15). The size of the buffer was stored in the data member IoStatus.Information of the IRP. Clearly, the `tag_id` 15 was modified in the tag trace graph. By examining the tag trace table, we found that the status of all http connections in the buffer were changed from 5 to 0.

5.3 Rustock.B

Rustock.B is a notorious backdoor rootkit that hides malicious activities on a compromised machine. The distinguished feature of this rootkit is the usage of multi-layered code packing, which makes static analysis cumbersome [7]. Unlike the other two rookits described above, we did not have access to the source code of this rootkit. However, several analysis results on this rootkit published on the Internet helped us understand some behaviors of this rootkit. We downloaded Rustock.B from [20] as one executable. During the test, we just double-clicked the binary and waited until the size of the Rkprofiler log stop being populated. The test lasted about five minutes.

A malicious driver named system32:lzx32:sys was detected by Rkprofiler. 90857 calls and 2936 tags were captured in the test. The driver contained self-modifiying code and we found many RET instructions that did not have corresponding CALL instructions at code unpacking stages. This is because unpacking routines executed JMP instructions to transfer the controls to the intermediate or unpacked code. In addition, the driver modified the dedicated register `IA32_SYSENTER_EIP` through WRMSR and RDMSR instructions to hijack the Windows System Service Descriptor Table (SSDT). One hook was added to the dispatch function table of driver Ntfs.sys to replace the original `IRP_MJ_CREATE` dispatch function. This is similar to what TCPIRPHOOK does. We compared the report generated by Rkprofiler with others on the Internet and they matched each other well. Table 2 lists the external functions and registry keys that were called and created by Rustock.B. Unfortunately, the full report of this test cannot be presented due to the space constraint.

6 Discussion

In addition to the incomplete kernel symbols provided by Microsoft, the current implementation of Rkprofiler suffers several other limitations that could be exploited by attackers to evade the inspection. First, attackers may compromise the kernel without running any malicious kernel code, e.g., directly modifying kernel data objects from user space or launching return-to-lib attacks without the use of any function calls [24]. Rkprofiler is not able to detect and profile such attacks. Instead, other defense approaches like control flow integrity enforcement [2] could be adopted to address them. Second, the instruction pair

Table 2. External functions and registry keys manipulated by Rustock. B

External Functions	ExAllocatePoolWithTag, ExFreePoolWithTag, ExInitializeNPagedLookasideList, IoAllocateMdl, IoGetCurrentProcess, IoGetDeviceObjectPointer, IoGetRelatedDeviceObject, KeClearEvent, KeDelayExecutionThread, KeEnterCriticalRegion, KeInitializeApc, KeInitializeEvent, KeInitializeMutex, KeInitializeSpinLock, KeInsertQueueApc, KeLeaveCriticalRegion, KeWaitForSingleObject, MmBuildMdlForNonPagedPool, MmMapLockedPages, MmProbeAndLockPages, NtSetInformationProcess, ObfDereferenceObject, ObReferenceObjectByHandle, ProbeForRead, PsCreateSystemThread, PsLookupProcessByProcessId, PsLookupThreadByThreadId, RtlInitUnicodeString, `_stricmp`, `_strnicmp`, swprintf, wcschr, wcscpy, `_wcsicmp`, `_wcslwr`, wcsncpy, `_wcsnicmp`, wcstombs, ZwClose, ZwCreateEvent, ZwCreateFile, ZwDeleteKey, ZwEnumerateKey, ZwOpenKey, ZwQueryInformationFile, ZwQueryInformationProcess, ZwQuerySystemInformation, ZwReadFile
Registry Keys	`HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\pe386` `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_pe386`

CALL/RET is used as the sole indicator of function call and return events. attackers can obfuscate these function activities to escape the monitoring. For example, JMP/JMP, CALL/JMP and JMP/RET can be employed to implement the function call and return events. Moreover, instead of jumping to a target instruction (either the first instruction of a callee function or the returned instruction of a caller function), a attacker could craft the code to jump to one of its neighbor instructions, while preserving the software logic intact. Defending against such attacks is part of our future work. Third, a attacker may deter the AMT method by accessing dynamic objects in unconventional ways. For example, a rootkit can scan the stack of a benign kernel function to get the pointer to a desired kernel object. These attacks are very challenging, because building an accurate and up-to-date symbol table for all kernel objects is impractical. Last, malware may have the capability of detecting virtual machine environments and change their behavior accordingly. Exploring multiple execution paths [16] and static analysis could mitigate this problem to some extent.

7 Related Work

Many previous works have focused on run time rootkit detection [19] [4] [17] [18] [28] and prevention [29] [22] [25]. The main purpose of these mechanisms is to protect data and code integrity of the guest OS at run time. On the other hand, researchers have also applied program analysis techniques to create offline rootkit defense mechanisms with goals such as rootkit identification, hook detection and so on. Several works that fall into this category are discussed below.

Rootkit Identification. Kruegel [13] proposed a system that performs static analysis of rootkits using symbolic execution, a technique that simulates program execution with symbols. This system can only detect known rootkits. Moreover, anti-static-analysis techniques like code obfuscation can be used to defeat this system. Limbo [26] is another rootkit analysis system that loads a suspicious driver into a PC emulator and uses flood emulation to explore multiple running paths of the driver. Limbo has a low false positive rate, but it performs poorly when detecting unknown rootkits. Also, flood emulation makes rootkits behave abnormally in the emulator, possibly resulting in inaccurate detection. Panorama [31] uses dynamic taint analysis to detect privacy-breaching rootkits. Sensitive system data like keys and packets are tainted and system-wide taint propagation is tracked. A taint graph is generated to tell whether a target rootkit accesses the tainted data or not. Although this system is good at capturing data-theft rootkits, it cannot provide necessary behavior information (e.g., kernel hooking) associated with other types of rootkits.

Hook Detection. HookFinder [30] and HookMap [27] aim to identify the hooking behavior of rootkits. HookFinder performs dynamic taint analysis and allows users to observe if one of the impacts (tainted data) is used to redirect the system execution into the malicious code. On the other hand, HookMap is intended to identify all potential hooks on the kernel-side execution paths of testing programs such as ls and netstat. Dynamic slicing is employed to identify all memory locations that can be altered to diverted kernel control flow. Unfortunately, hooking is only one aspect of rootkit behavior and both systems cannot provide comprehensive a view of rootkit activities in a compromised system.

Discovery of Sensitive Kernel Data Manipulation. K-tracer [14] is a rootkit analysis system that automatically discovers the kernel data manipulation behaviors of rootkits including sensitive data access, modification and triggers. K-tracer performs data slicing and chopping on sensitive data in the rootkit trace and identifies the data manipulation behaviors. K-tracer cannot detect hooking behaviors of rootkits and is unable to deal with DKOM and hardware-based rootkits. In comparison, Rkprofiler can handle a broad range of rootkits, including DKOM and hardware-based rootkits, and provide a complete picture of rootkit activities in a compromised system.

Rootkit Profiling. PoKeR [23] is a QEMU-based analysis system that shares the same design goal as Rkprofiler: comprehensively revealing rootkit behavior. PoKeR is capable of producing rootkit traces in four aspects: hooking behavior, target kernel objects, user-level impact and injected code. Similar to Rkprofier, PoKeR infers the dynamic kernel object starting from the static kernel objects. However, PoKeR only tracks the pointer-based object propagation, while Rkprofiler tracks both pointer-based and function-based object propagation. So Rkprofiler can identify more kernel objects than PoKeR. In addition, the function call and hardware access monitoring features of Rkprofiler are not offered by PoKeR.

8 Conclusion

In this paper, we present a sandbox-based rootkit analysis system that monitors and reports rootkit behavior in a guest OS. The evaluation results demonstrate the effectiveness of this system in revealing rootkit behavior. However, to strengthen the current implementation of Rkprofiler, we need OS vendors to provide the unpublished symbols, some of which may have been reversely engineered by attackers.

References

1. Anubis Project (2009), http://anubis.iseclab.org/?action=home
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, j.: Control-flow integiryt: Principles, implementations, and applications. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2005)
3. BitBlaze Project (2009), http://bitblaze.cs.berkeley.edu/
4. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proceedings of the 24th Annual Computer Security Applications Conference, ACSAC (2008)
5. Bellard, F.: QEMU and Kqemu (2009), http://fabrice.bellard.free.fr/qemu/
6. CBS News. Conficker Wakes Up (2009), http://www.cbsnews.com/stories/2009/04/09/tech/cnettechnews/main4931360.shtml
7. Chiang, K., Lloyd, L.: A case Study of the Rustock Rootkit and Spam Bot. In: First workshop on hot topics in understanding botnets (2007)
8. Dr.Web Company. Win32.Ntldrbot (aka Rustock.C) no longer a myth, no longer a threat. New Dr.Web scanner detects and cures it for real (2009), http://info.drweb.com/show/3342/en
9. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the Symposium on Network and Distributed System Security, NDSS (2003)
10. Geeg Blog. The Conficker Worm Awakens (2009), http://geeg.info/blog4.php/2009/04/the-conficker-worm-awakens
11. GraphViz Project (2009), http://www.graphviz.org/
12. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2005)
13. Kruegel, B.C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits through Binary Analysis. In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC (2004)
14. Lanzi, A., Sharif, M., Lee, W.: K-Tracer: A System for Extracting Kernel Malware Behavior. In: Proceeding of the Annual Network and distributed System Security Symposium, NDSS (2009)
15. Microsoft Symbol Server (2009), http://msdl.microsoft.com/download/symbols
16. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the IEEE Symposium on Security and Privacy (2007)
17. Petroni, N.L., Fraser, T., Molinz, J., Arbaugh, W.A.: Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In: Proceedings of the USENIX Security Symposium (2004)

18. Petroni, N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proceedings of the USENIX Security Symposium (2006)
19. Petroni, N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2007)
20. Offensivecomputing Website (2009), http://www.offensivecomputing.net/
21. Rootkit website (2009), http://www.rootkit.com
22. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
23. Riley, R., Jiang, X., Xu, D.: Multi-Aspect Profiling of Kernel Rootkit Behavior. In: Proceedings of the ACM SIGOPS European Conference on Computer Systems, EuroSys (2009)
24. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2007)
25. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP (2007)
26. Wilhelm, J., Chiueh, T.: A Forced Sampled Execution Approach to Kernel Rootkit Identification. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 219–235. Springer, Heidelberg (2007)
27. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering Persistent Kernel Rootkits Through systematic Hook Discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
28. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Recontruction. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2007)
29. Xuan, C., Copeland, J., Beyah, R.: Shepherding Loadable Kernel Modules through On-demand Emulation. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 48–67. Springer, Heidelberg (2009)
30. Yin, H., Liang, Z., Song, D.: Hookfinder: Identifying and understanding malware hooking behaviors. In: Proceeding of the Annual Network and distributed System Security Symposium, NDSS (2008)
31. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Captureing System-wide Information Flow for Malware Detection and Analysis. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (2007)

Exploiting Temporal Persistence to Detect Covert Botnet Channels

Frederic Giroire[1], Jaideep Chandrashekar[2], Nina Taft[2], Eve Schooler[2], and Dina Papagiannaki[2]

[1] Project Mascotte, I3S(CNRS/UNS)/INRIA
[2] Intel Research

Abstract. We describe a method to detect botnet command and control traffic and individual end-hosts. We introduce the notion of "destination traffic atoms" which aggregate the destinations and services that are communicated with. We then compute the "persistence", which is a measure of temporal regularity and that we propose in this paper, for individual destination atoms. Very persistent destination atoms are added to a host's whitelist during a training period. Subsequently, we track the persistence of new destination atoms not already whitelisted in order to identify suspicious C&C destinations. A particularly novel aspect is that we track persistence at multiple timescales concurrently. Importantly, our method does not require any a-priori information about destinations, ports, or protocols used by the C&C communication, nor do we require payload inspection. We evaluate our system using extensive user traffic traces collected from an enterprise network, along with collected botnet traces.

We demonstrate that our method correctly identifies a botnet's C&C traffic, even when it is very stealthy. We also show that filtering outgoing traffic with the constructed whitelists dramatically improves the performance of traditional anomaly detectors. Finally, we show that the C&C detection can be achieved with a very low false positive rate.

1 Introduction

A botnet is a collection of compromised end-hosts all under the control of a particular *bot-master* (or *bot-herder*). The recruited end-hosts, also called *drones* or *zombies*, are marshalled and controlled by the bot-herders via a command and control (in short, C&C) traffic channel to carry out a number of malevolent activities. For example, they are used to launch DDoS attacks, send SPAM, harvest personal information from zombie hosts, stage social engineering attacks, and so on. Botnets are so effective at delivering these services that there is an thriving (underground) economy based around buying or renting botnets [1]. Today's commercial malware prevention methods, typically host based HIPS and AV engines, are well suited to identifying and countering previously identified and analyzed threats. However, contemporary botnets are extremely adaptable and able to churn our variants at a very high volume, using polymorphism and

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 326–345, 2009.
© Springer-Verlag Berlin Heidelberg 2009

packing engines, which can easily overwhelm existing defenses (a particular AV vendor reports collecting 3000 distinct malware samples daily on average [2]).

In contrast to signature scanning based methods, which target known threats, statistical anomaly detection methods are often employed to detect new threats; these operate by looking for deviations in traffic feature distributions caused by the malware. These methods can detect and flag zombie hosts that have been activated and generating a significant (noticeable) volume of traffic (DDoS attacks, SPAM, click-fraud, etc). However, it may be a considerable period of time between a host joining a botnet to the time that is instructed to carry out a malicious task; often by then it is too late, as the zombie has completed its purpose. Therefore, even as detecting a botnet in the act of performing some detrimental activity should be a goal, it is far more critical to block the initial recruitment vector, or failing that to detect the C&C traffic between the drone and bot-herder, so as to deactivate the channel and render the drone useless. More critically, information gathered about the C&C infrastructure may be used to take down the botnet as a whole.

In this paper, we present and validate a method to detect the C&C communications at an endhost. We were motivated by the observation that a recruited host needs to be in touch with its C&C server to be ready to carry any particular activity. It will reconnect, for each new activity, each time it is repurposed, or resold, and so on. Intuition suggests that such visits will happen with some regularity; indeed without frequent communication to a C&C server, the bot becomes invisible to the bot herder. However, this communication is likely to be very lightweight and spaced out over irregular large time periods. This helps the botnet be stealthy and hence harder to expose. We thus seek to design a detector that monitors a user's outgoing traffic in order to expose malicious destinations that he visits with some temporal regularity, even if infrequently. In order to discern these from normal destinations a user frequents, we build whitelists based on a new kind of IP destination address aggregation which we call *destination atoms*. A destination atom is an aggregation of destinations that is intended to capture the service connected to. For example, we view *google.com* as a destination service, because a user's request to *google.com* will be answered, over time, by many different servers with different IP addresses. We build these destination atoms using a series of heuristics. Then, to capture the nebulous idea of "lightweight repetition", we introduce a measure called *persistence*. Our whitelists contain destination atoms that exhibit a given level of persistence. With this whitelist in place, detection proceeds by tracking persistence to contacted (non whitelisted) destinations. When the computed persistence becomes high enough, the destination is flagged as a potential C&C endpoint. The method we describe in this paper is particularly well suited to be deployed inside enterprise networks. In such networks, a level of administrative control is enforced over what applications may or may not be installed by end-users, which results in traffic from individual end-hosts being easier to analyze and attribute to particular applications.

The regularity with which a zombie contacts its bot-herder will differ from botnet to botnet; moreover, we cannot predict the communication frequency that will be used in tomorrow's botnet. We therefore propose to track persistence over multiple timescales simultaneously so as to expose a wide variety of communication patterns. We develop a simple and practical mechanism to track persistence over many observation windows at the same time.

There are various styles by which botnets can communicate to command control centers, including IRC channels, P2P overlays, centralized and decentralized C&C channels. Our goal here is to try to uncover the C&C activity for the class of bots that employ a high degree of centralization in their infrastructure and where the communication channel lasts for an extended period. We do not target botnets where the communication between zombie and bot-herder is limited to a few connections, or those where the zombie is programmed to use a completely new C&C server at each new attempt.

We validate and assess our scheme using two datasets; one consists of (clean) traces collected directly on enterprise endhosts, and the second consists of traces of live bot malware. For our method to be practical, it is important that the created whitelists be stable, i.e., they do not require frequent updating. In addition, it is essential that whitelists be small so that they require little storage and can be matched against quickly. Using data traces from a large corpus of enterprise users, we will show that the constructed whitelists, composed of destination atoms and based on their persistence values, exhibit both of these properties. We then overlay the malware traces on top of the user traces, and run our detectors over this combined traffic trace. We manually extracted the C&C traffic from the botnet malware traces in order to compute false positives and false negatives. We show that our method identifies the C&C traffic in all the bots we tested. Finally, we also demonstrate that there is a positive side benefit, of identifying the persistent destination atoms. The sensitivity of HIDS traffic anomaly detectors can be dramatically improved, by first filtering the traffic through the whitelists. This allows a larger fraction of our endhosts to catch the attack traffic, while also speeding up the overall detection time.

2 Related Work

There are three potential avenues with which we could mitigate the botnet problem as described in [3]: preventing the recruitment, detecting the covert C&C channel, and detecting attacks being launched from the (activated) drones. A number of previous works has addressed the first avenue ([4,5] among others), and in this paper we chiefly address the second avenue (and the third, albeit indirectly). Our method detects the covert channel end-points by tracking persistence, and we are able to detect attacks by filtering out whitelisted (normally persistent) traffic and subsequently applying well established thresholding methods, borrowing from the domain of statistical anomaly detection.

In [6] the authors devise a method to detect covert channel communications carried over IRC with a scoring metric aimed at differentiating normal IRC channels from those used by botnets based on counts of protocol flags and common

IRC commands. Our own work differs in that we do not require protocol payloads, nor are we restricted to IRC activity. Another detection approach, BotHunter [7], chains together various alarms that correspond to different phases of a host being part of a botnet Our own method does not attempt to identify such causality and is also able to detect botnet instances for which other identifying alarms do not exist. Other approaches to detecting botnet traffic involve correlating traffic patterns to a given destination, across a group of users [8]. Our own work is complementary, focusing on the individual end-host (and can thus detect single instances of zombies); we can envision combining the approaches together. Botminer [9] attacks the detection problem by first independently clustering presumed malicious traffic and normal traffic and then performing a cross-correlation across these to identify hosts that undertake both kinds of communication. These hosts are likely to be part of an organized botnet. Our own work differs in that it is primarily an end-host based solution, and we do not attempt to correlate activities across hosts. Also, we do not attempt to identify attack traffic in the traffic stream. We focus purely on the nature of communication between the end-hosts and purported C&C destinations, looking for regularity in this communication. Orthogonal to the problem of detecting the botnets and their activities, and following the increasing sophistication being applied in the design of the more common botnets in operation today, there has been a great deal of interest in characterizing their structure, organization and operation. A description of the inner workings, specifically, the mechanisms used by the Storm botnet to run SPAM campaigns is described in [10]. The work in [11] examines the workings of *fast-flux* service networks, which are becoming more common today as a way to improve the robustness of botnet C&C infrastructures.

The area of traffic anomaly detection is fairly well established, and many detectors have been proposed in the past [12,13]. Some methods build models of normal behavior for particular protocols and correlate with observed traffic. An indicator of abnormal traffic behavior, often found in scanning behaviors, is a unusual number of connection attempts that fail, as detailed in [12]. Another interesting idea, described in [13], in which the authors try to identify the particular flow that caused the infection by analyzing patterns of communications traffic from a set of hosts simultaneously. All of these approaches are complementary to our work and we allow for any type of traffic feature based anomaly detector to be integrated into the system we describe. Finally, [14] describes a method to build host profiles based on communication patterns of outgoing traffic where the profiles are used to detect the spread of worms. This is different from our goal in this paper, to detect botnet C&C activity, which is a stealthier phenomenon. A more fundamental difference between our approaches is that we employ the notion of persistence to incorporate temporal information.

To end this discussion we strongly believe, given the severity of the issue, there is not a single silver bullet solution that can tackle the botnet threat; a combination of mechanisms will be needed to effectively mitigate the problem. We view our own work as complementary to existing approaches that are focused on preventing the botnet recruitment or else protecting against the infection vector.

3 Methodology

While botnets vary a great deal in organization and operation, a common behavior across all of them is that each zombie needs to communicate regularly with a C&C server to verify its liveness. In order to keep the C&C traffic under the radar, most botnets tend to keep this communication very lightweight, or stealthy. However because the bot will visit its C&C server repeatedly over time, failing which the bot-herder might simply assume the zombie to be inactive, we are motivated to try to expose this low frequency event. To do this, we introduce a notion called *destination atoms* (an aggregation of destinations), and a metric called *persistence* to capture this "lightweight" yet "regular" communication. We design a C&C detection method that is based upon tracking the persistence of destination atoms. In order to differentiate whether a new destination atom exhibiting persistence is malicious or benign, we need to develop whitelists of persistent destinations that the user or his legitimate applications normally visit.

The intuition for our method is as follows: an end-host, on any particular day, may communicate with a large set of destination end-points. However, most of these destinations are transient; they are communicated with a few times and never again. When traffic from the host is tracked over longer periods, the set of destinations visited regularly is a (much) smaller and stable set. Presumably, this set consists of sites that the user visits often (such as work related, news and entertainment websites), as well as sites contacted by end-host applications (such as mail servers, blogs, news sites, patch servers, RSS feeds, and so on). If the set of destinations with high regularity is not very dynamic, then such a set can define a behavioral signature of the end-host and can be captured in a whitelist that requires infrequent updating (once learned). We will see in our user study, that indeed such a whitelist is quite stable. This means that should a new destination appear, one that is persistent and malicious, the event stands out, enabling detection. This is precisely what we expect to happen when an end-host is subverted, recruited into a botnet and begins to communicate with its C&C servers.

In order to keep the whitelists compact and more meaningful, we use a set of heuristics to aggregate individual destination endpoints into *destination atoms*, which are logical destinations or services. For example, the particular addresses that are mapped to `google.com` vary by location and time, but this is irrelevant to the end user who really only cares about the `google` "service". The same is often true for mail servers, print services, and so on. For our purpose, we primarily care about the network *service* being connected to, not so much the actual destination address.

Given a destination end-point (dst**IP**, dstPort, proto), we obtain the atom (dst**Service**, dstPort, proto), by extracting the *service* from the IP address using the heuristics described below (Table 1 shows a few mappings from endpoints to destination atoms):

1. If the source and destination belong to different domains, the service name is simply the second level domain name of the destination (e.g., cisco.com, yahoo.com)

Table 1. Example destination atoms contacted by `somehost.intel.com`. Notice that the `intel` hosts, being in the same domain, are mapped onto the third level domain, and the `google` destinations to the second level domain.

Destination address	Dest. Name	Dest. Atom
(143.183.10.12, 80, tcp)	www.inet.intel.com	(inet.intel.com, 80, tcp)
(134.231.12.19, 25, tcp)	smtp-gw.intel.com	(smtp-gw.intel.com, 25, tcp)
(216.239.57.97, 80, tcp)	cw-in-f97.google.com	(google.com, 80,tcp)
(209.85.137.104, 80, tcp)	mg-in-f104.google.com	(google.com, 80,tcp)

2. If the source and destination belong to the same domain, then the service is the third level domain name (e.g., mail.intel.com, print.intel.com). We differentiate these situations because we expect a host to communicate with a larger set of destinations in its own domain, as would be the case in an enterprise network.
3. When higher level application semantics are available (such as in enterprise IT departments), we can use the following type of heuristic. Consider the passive FTP service, which requires two ports on the destination host, one being port 21, and the other an ephemeral port (say k). Thus, both (ftp.service.com,21,tcp) and (ftp.service.com,k,tcp) reflect the *same service* and for the case of FTP. We thus generalize the destination atom as (ftp.service,com, 21:>1024,tcp). Being able to do this for a larger set of protocols requires a detailed understanding of the particular application's semantics, which is beyond our scope here. In this paper, we use a simple heuristic that approximates this behavior: if we observe more than 50 ephemeral ports being connected to at the same service, we expand the destination atom to include all ephemeral ports. The rationale here is that if the service is associated with ephemeral ports, it is likely that we will observe a port number not seen previously at some time and should simply include this in the previously created destination atom.
4. Sometimes a single destination host can provide a number of distinct services, and in this case, the destination port is sufficient to disambiguate the services from each other, even though they may have similar "service names", obtained by a (reverse) DNS lookup.
5. Finally, when the addresses cannot be mapped to names, no summarization is possible, and we use the destination IP address as the service name.

We now define our persistence metric, to quantify the lightweight yet regular communication from end-hosts to destination atoms. We monitor the outgoing traffic from an end-host using a large sliding time window of size W, divided into n time-slots, each of length s. We term W an *observation window*, and each time-slot s as a *measurement window* (simply, *bin*). Letting s_i denote the i-th slot of size s in W, we have $W \equiv [s_1, s_2, \ldots, s_n]$. The persistence of a destination atom (`d`), in the observation window W is defined as:

$$p(\mathtt{d}, W) = \frac{1}{n} \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathbf{1}_{\mathtt{d},\mathrm{s_i}}$$

where $\mathbf{1}_{\mathtt{d},\mathrm{s_i}}$ has a value 1 if the host makes at least one connection to d in the time-slot s_i, and 0 otherwise. Thus, a destination atom's persistence value is simply the fraction of time slots where at least one connection was observed. Given a threshold p^*, we say that d is *persistent* if $p(\mathtt{d}, W) > p^*$ (otherwise, it is termed *transient*).

Because botnets differ from one to another to a great extent, we cannot know a priori the frequency (using the term loosely) with which a zombie will contact its C&C server(s). Thus, it is of paramount importance to design a method that can track persistence over several observation windows simultaneously. Note that the persistence of an atom depends upon the sizes of the two windows (W, s) over which it is observed and measured; we use the term *timescale* to denote a particular instance of (W, s). In order to capture persistence over many timescales, we select k overlapping timescales $(W^1, s^1) \subset (W^2, s^2) \subset \ldots \subset (W^k, s^k)$, where (W^1, s^1) is the smallest timescale, and (W^k, s^k) is the largest. Here s^j denotes the bin size at time scale j. (We could define s_i^j as the ith slot in an observation window W^j, however we drop the subscript for simplicity when discussing timescales). For each timescale $(W^j, s^j) : 1 \leq j \leq k$, we compute the persistence $p^{(j)}(\mathtt{d})$ as previously defined. Then, a destination atom d is *persistent* if the threshold p^* is exceeded in *any one* of the timescales, i.e., d is a *p*ersistent destination atom iff

$$\max_{\mathrm{j}} p^{(j)}(\mathtt{d}) > p^*$$

We have explicitly chosen not to use direct frequency type measurements (e.g., a low pass filter) because our definition is very flexible and does not require one to track specific multiples of one frequency or another. More importantly, we don't expect these low frequency connection events to precisely align at any particular frequency; our definition allows some slack in where exactly the events occur.

When deciding upon the appropriate timescales, particularly the smallest measurement window s^1, we want it capture session level behavior (multiple requests to a web server in a short interval are likely to be for a single session). Based on a preliminary analysis of user data, we select $s^1 = 1$ hr (from empirical data, we see that 87% of connections to the same destination atom are separated by at least an hour). We also set $s^k = 24$ hours because our training dataset is 2 weeks (in reality, one could use larger windows). With these two boundary slot-lengths, we select a number of intermediate values (listed in § 5.2). The observation window length controls how long we should wait before we conclude as to whether something is persistent (or transient). For convenience, we select $n = 10$. Noting that $W = n \times s$, the 7 timescales used in this paper lie between $(W^{min} = 10, s^{min} = 1)$, which is the smallest timescale, and $(W^{max} = 240, s^{max} = 24)$, the largest (all values described in hours). It should be pointed out that additional timescales can be added dynamically based on evidence of some anomaly at a particular timescale.

In the following, we describe the specifics of how our C&C detection proceeds. Initially, there is a training stage in which the end-host bootstraps itself by learning a whitelist of destination atoms. The training stage should last long enough for the stable behavior of the end-host to be exposed (perhaps a week or two). Subsequent to the training stage, after the whitelists are built up and system parameters initialized, the system enters the detection stage. It should be noted that the training and detection stages proceed identically; in both, persistence of destinations is tracked and alarms raised when this crosses a specified threshold. The fundamental difference is that in the detection stage, an alarm simply results in the destination atom being incorporated into the whitelist; on the other hand, in the detection stage, the alarm alerts the end-user (or communicated to the central IT console in an enterprise) and asked to take punitive action. In case the alarm is benign, and the user (or administrator) can attest to the destination atom, it is added into the whitelist.

C&C Detection Implementation. To simplify the description, we first describe how detection is carried out with a single timescale. The system proceeds by observing all the outgoing traffic at an end-host; connections to destination atoms already in the whitelist are ignored. Persistence values are tracked for all other outgoing traffic: connections to a destination atom in a window W, is tracked using a bitmap of n bits (one bit for each timeslot in $s \in W$). If a new outgoing connection is observed in slot s_i, then the entry in the bitmap, for the corresponding slot, is set to 1. We create a separate bitmap for each destination atom when it is first observed. This bitmap updating occurs asynchronously as the outgoing connections are observed. A timer fires every s^{min} minutes (this is the time interval corresponding to the smallest timescale), and all the bitmaps are processed. Here, for each bitmap, the persistence is computed taking into account the last n bits and at this time, one of three things can occur: (i) it cannot be determined if the persistence is high enough (not enough samples), and the bitmap is retained; (ii) the newly updated persistence crosses the critical threshold, p^* (bitmap is freed and an alarm is raised), or (iii) after enough samples, the persistence is below the threshold (the bitmap is freed up).

In order to track persistence at multiple timescales simultaneously, we could use k separate bitmaps per atom. It turns out this is not necessary because we can exploit the structure in our definition of timescales to reduce the overhead. Notice that $s^1 \cdot n = W^1 < W^2 < \ldots < W^k = n \cdot s^k$, that is, s^j is covered by a slot in the next higher timescale, as is depicted in Fig. 1. Thus, setting a bit in one of these timescales implies setting a bit in the higher timescale. Thus, rather that maintain separate bitmaps, we can simply construct a single, *long* bitmap that covers all the timescales appropriately; this allows all the timescales to be updated with a single bit update. The length of this bitmap is $\frac{W^k}{s^1} = \frac{W^{max}}{s^{min}}$. In our implementation we have $s^1 = 1\ hr$, $n = 10$, and $W^{max} = 240$ hrs, so the bitmap length is exactly 240 bits.

High level pseudocode for this entire process is shown in Proc. 1. Here, the set of bitmaps is stored in `DCT`, indexed by individual atoms (line 1 retrieves all

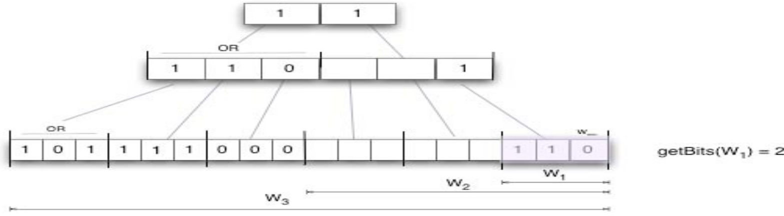


Fig. 1. Bitmaps to track connections at each timescale. Here, we have $n = 3$ and $k = 3$.

Algorithm 1. computePersistence():

1: **for all** $\mathtt{d} \in \mathtt{DCT}$ **do**
2: $p(\mathtt{d}) \leftarrow 0$
3: **for** $i = 1$ to k **do**
4: $p^{(i)}(\mathtt{d}) \leftarrow \mathsf{getBits}(\mathtt{d}, i).\frac{|W^i|}{|s^i|}$
5: $p(\mathtt{d}) = \mathsf{max}(p(\mathtt{d}), p^{(i)}(\mathtt{d}))$
6: **if** $p(\mathtt{d}) \geq p^*$ **then**
7: RAISEALARM(... suspicious destination d)
8: **end if**
9: **end for**
10: $\text{idx} \leftarrow (\text{idx} + 1)\text{mod}\frac{W^{\max}}{s^{\min}}$
11: **if** $p(\mathtt{d}) = 0$ **then**
12: discard DCT[d]
13: **end if**
14: **end for**

the active bitmaps). The loop (lines 2-7) iterates over each timescale, computing persistence in each. There is a separate process that processes each outgoing connection; this checks if the destination is whitelisted, and if not, updates the bit at index idx in the bitmap (this index is updated in line 10, each time the procedure is called, i.e., every s^{min}). Finally, if there is no observed activity for the atom in W^{max}, the bitmap is discarded (lines 11-13).

When a C&C alarm is raised, we flag the outgoing connection as suspicious and alert the user who can choose between either adding the destination atom to their whitelist, or blocking the outgoing traffic. We will see in our evaluation that the users are bothered with such decisions infrequently. For enterprise users, such alarms could also be passed to an IT department where they can be correlated with other data.

4 Dataset Description

End Host Traffic Traces. The endhost dataset used in this paper consists of traffic traces collected at over 350 enterprise users' hosts (mostly laptops), over a 5 week period. Users were recruited via intranet mailing lists and newsletters,

and prizes were offered as a way to incentivize participation. The results presented in this paper use the traces from 157 of the hosts; these were selected because they provide trace data for a common 4 week period between January and February 2007. Our monitoring tool collected all packets headers, both ingoing and outgoing, from all machine interfaces (wired and wireless). We divide the 4 weeks traffic trace into two halves, a training set and a testing set. The training data is used to build the per-user whitelists and to determine the system parameters (i.e. p^*). The testing data is used to assess the detection performance, i.e., false positives and false negatives.

Botnet Traffic Traces. We collected 55 botnet binaries randomly selected from a larger corpus of malware. Each binary was executed inside a Windows XP SP2 virtual machine and run for as long as a week, together with our trace collection tool. When constructing the clean VM image, we took great pains to turn off all the services that are known to generate traffic (windows auto-update, etc.) This gives us a certain level of confidence that all (or nearly all) the traffic collected corresponds to botnet activity. During the collection, the server hosting the VMs was placed behind a NAT device and connected to an active DSL link.

While we expected the trace collection to be a straight-forward exercise, this turned out not to be the case. To begin with, a lot of the binaries simply crashed the VM or else did nothing (no traffic was observed). In other cases, the C&C seemed to have been deactivated, and we only saw failed connections or connections to illegal addresses (indicating that the DNS entries for the C&C servers had been rewired). Only 27 binaries yielded any traffic at all (the collection was aborted if there was no traffic seen even after 48 hours). From the larger set of 55 , only 12 binaries yielded traffic that lasted more than a day and the details of these (usable) traces are enumerated in the first column of Table 2. The labels here are obtained using the open source ClamAV scanning engine [15]. Given our limited infrastructure, we were unable to scale to a large number of binaries; however, we endeavored to collect botnet samples with widely different behaviors and temporal characteristics to assure us that our evaluation results hold for a larger population of botnets.

To evaluate our detection method, we overlaid these botnet traces on the traffic traces from each user, and then run this combined traffic through our detector which tries to identify persistent destination atoms. In order to assess the true detections, missed detections and false positives, we first needed to obtain the ground truth from the botnet traces (for which there are no established methods). We had to manually analyze all of our 12 botnet traces in order to isolate the C&C traffic from the remainder of the attack traffic. Rather than label individual packets we used BRO [16] to generate connection summaries which we analyzed in detail.

Isolating the C&C traffic turned out to be a very tedious process which involved manually breaking down the traffic across ports and destinations of each botnet trace. Due to a lack of space, we cannot enumerate how we did this for the entire set. In summary, we employed a variety of methods including, extracting IRC commands embedded in the streams, looking at the payloads directly

Table 2. List of sampled Botnet binaries with clear identifiable C&C traffic

ClamAV Signature	C&C type	# of C&C atoms	C&C Volume min - max
Trojan.Aimbot-25	port 22	1	0-5.7
Trojan.Wootbot-247	IRC port 12347	4	0-6.8
Trojan.Gobot.T	IRC port 66659	1	0.2-2.1
Trojan.Codbot-14	IRC port 6667	2	0-9.2
Trojan.Aimbot-5	IRC via http proxy	3	0-10
Trojan.IRCBot-776*	HTTP	16	0-1.
Trojan.VB-666*	IRC port 6667	1	0-1.3
Trojan.IRC-Script-50	IRC ports 6662-6669,9999,7000	8	0-2.1 8
Trojan.Spybot-248	port 9305	4	3.8-4.6
Trojan.MyBot-8926	IRC port 7007	1	0-0.1
Trojan.IRC.Zapchast-11	IRC ports 6666, 6667	9	0-1
Trojan.Peed-69 [Storm]	P2P/Overnet	19672	0-30

of non-IRC communications, and in some cases examining histograms of payload size to extract unusual patterns (i.e. very high chance of small packet sizes consistent across a subset of connections). As an interesting example, consider Trojan.AimBot-5. First we constructed histograms of traffic to various destinations and on various ports. In this particular case, the communication involved a few destinations. By zooming in on these individual connections and reconstructing the associated TCP streams we obtained a "conversation" between the zombie and the significant destination. We were able to identify IRC protocol commands being tunneled over HTTP to particular destinations. Further analysis revealed that the destination being contacted was hosting a squid proxy, and the IRC commands were being tunneled through. In case of the Storm botnet trace, we were able to pick out the p2p traffic from the packet size distributions (the UDP traffic used to communicate with the p2p network had a very different characteristic from the other, presumably attack, traffic). The second column in Table 2 describes the ports and protocols associated with the C&C channel. The third column is a count of the distinct destination atoms seen in the (isolated) C&C traffic. Column 4 shows the range (min to max) of C&C traffic in connections/minute. This confirms our belief that the communication volume associated with the C&C traffic is light and thus volume based detectors would not be able to easily expose this traffic.

5 Evaluation

In this section we present results from overlaying the botnet malware traces on top of each user trace, and then emulating our detection algorithm to evaluate the performance of our detector. The two notable results which we discuss further in this section are: (i) our persistence metric based detector can indeed pick out the C&C destination atoms in the botnet traces with a very low false positive rate, (ii) the whitelists we construct can significantly boost the detection rates, and improve detection times, of well established volume anomaly detectors.

5.1 System Properties

As mentioned earlier, for our system to work well, the whitelists should have two properties. First, they should be stable, i.e., they need to be updated infrequently (so that bot C&C will stand out and user annoyance is kept small). Second, it is desirable that they be small, as this speeds up the searching activity (matching outgoing traffic against whitelist contents). Our whitelists will be stable if the rate at which new persistent destination atoms are added to the whitelist is low; and this will be true when much of the user communication is transient. To examine this for our set of users, we compute all the destination atoms for a given user and the persistence value for each atom. The cumulative distribution of these across all users is plotted in Figure 2. We see that less than 20% of the destination atoms have a persistence value greater than 0.2; this validates our intuition that transient destinations form the bulk of endpoints a host communicates with. Very few destination atoms exhibit a persistence greater than 70%. The observation that any user typically has few persistent destination atoms confirms that a whitelist consisting of these atoms is unlikely to be updated often. Recall that our method uses a parameter p^*, that is used both to construct the whitelists in the training stage, and as an alert threshold when monitoring for new C&C destination during detection (testing phase). This plot of user data suggests that selecting a value of p^* anywhere in the range of 50 to 80% will result in a small whitelist, that is likely to require few updates. We select the value of $p^* = 0.6$ because it is in the flat portion of the curve. Note that the number of destination atoms in the whitelist is not very sensitive to the value of p^* (as long as it is above roughly 0.5) suggesting that this parameter is fairly robust in the sense that it need not necessarily be fine tuned. In figure 3, we plot the histogram of whitelist sizes across all the 157 users. The whitelists for almost all the hosts contain 60-140 destination atoms, which is a very manageable size and thus limits any overhead involved in searching whitelists when filtering. Thus our user traffic traces confirms that whitelists constructed of persistent destination atoms will have the two desirable properties which will enable our detection method to be effective.

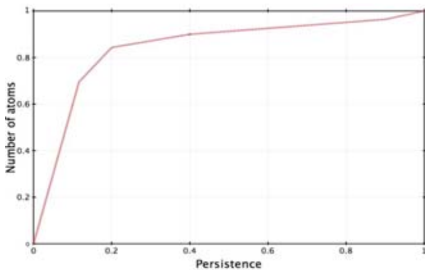


Fig. 2. CDF of destination atom persistence across all atoms seen in training data

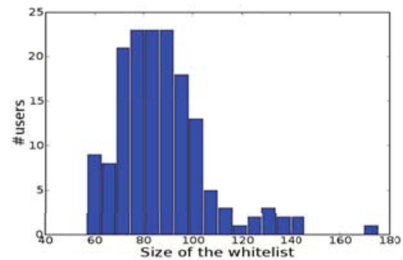


Fig. 3. Distribution of per host whitelist size (using $p^* = 0.6$)

Ideally, we would have liked to verify these properties over a much longer period. However collecting user traces, particularly over an extended period and a sizeable population is extremely challenging. Nor are there public repositories of such data that can be used. While we cannot prove this conclusively, there are certain observations to be made from the traces we use which lead us to believe that the properties are likely to hold over a longer period. One of these is that we see few false positives in the C&C detection (§̃refsec:sec:ccdetection), which essentially determines how the whitelist sizes will grow over time.

5.2 C&C Detection

To assess the ability of our algorithm to identify C&C traffic when it is mixed in with regular user traffic, we overlaid each botnet trace on each of our 157 user traces and ran this traffic through our detector (12 botnets $\times$ 157 users = 1884 traces). The detector was configured to use 5 distinct timescales (as discussed in Section 3). The timescales used were with measurement window s taking values $s = (1, 4, 8, 16, 20, 24)$, and the observation window W was always $W = 10s$, i.e. we used (1,10), (4,40), etc. In each of these 1884 instances, our detector was able to correctly identify the C&C traffic. This was validated against the labels determined earlier (from having isolated the portion of the bot traffic corresponding to the C&C channel). This success illustrates the effectiveness of our persistence metric in detecting botnet C&C activity.

In Table 3 we list various properties of the detected botnets. Column 2 indicates the persistence of a destination atom from a particular bot. Column 3 indicates the timescale that triggered the alert, and the 4th column enumerates the specific number of destination atoms that were associated with (persistence and timescale) listed in the same row. For example, we see IRCBot-776 listed twice (first two rows of this table) because it used one destination atom that had a persistence of 1 and was detected at a timescale of (10,1), and it had 2 other destination atoms with a persistence of 0.8 that were detected at a timescale of (200,20). This example illustrates that a single zombie might used multiple time scales (in terms of how regularly they contact their botmasters) on different C&C servers. Looking down column 3, we see that the smallest timescale (10,1) was sufficient to detect at least one of the atoms in all instances except in the case of IRC.Zapchast-11 and Mybot-8926. However, we cannot know ahead of time as to what timescale is appropriate for a particular botnet; thus it is critical to have enough timescales in play to cover a wide range of behaviors. For the STORM bot, we have marked "> 1" in the last column because there a great many of them. The success of our method in uncovering the STORM bot C&C traffic brings up an interesting point: even though our method works best to uncover botnets that tend to have a high degree of centralization, we are able to detect the p2p based infrastructure used by the Storm botnet. Thus, our method is likely to be effective at also uncovering non-centralized infrastructures as long as there is a certain repetitiveness in contacting some of the destination atoms involved (out of the thousands, in the case of Storm).

Table 3. C&C Detection Performance

Botnet	Persistence	Timescale	# dest. atoms
IRCBot-776	1.0	(10,1)	1
IRCBot-776	0.8	(200,20)	2
Aimbot-5	1.0	(10,1)	1
Aimbot-5	1.0	(40,4)	1
Aimbot-5	1.0	(160,16)	1
MyBot-8926	0.6	(160,16)	1
IRC.Zapchast-11	1.0	(40,4)	3
Spybot-248	1.0	(10,1)	2
IRC-Script-50	1.0	(10,1)	7
VB-666	0.7	(10,1)	1
Codbot-14	1.0	(10,1)	1
Gobot.T	1.0	(10,1)	1
Wootbot-247	1.0	(10,1)	3
IRC.Zapchast-11	1.0	(10,1)	6
Aimbot-25	1.0	(10,1)	1
Peed-69 [Storm]	1.0	(10,1)	> 1

The bot `IRC.Zapchast-11` presents a compelling illustration on how tracking for persistence can be effective even when the connection volume is extremely stealthy. Recall from Table 2 that `IRC.Zapchast-11` generates very little traffic overall - about 1.4 connections per binning interval on average. By all accounts, this is a minuscule amount of additional traffic, that has no chance to stand out in traffic volume against the normal traffic of an end-host, and thus will go undetected by a volume based anomaly detector. However by tracking the persistence of its associated destination atom, we were able to make the anomaly visible. This illustrates the utility of persistence, even in the face of extremely stealthy bots.

Using our two data sets, we can also compute the false positive and detection rate (1 - false negative rate) tradeoff. We computed a traditional ROC curve, by sweeping through a range of values for p^*. The detection rate is computed as the fraction of the tested botnets, across users, for which an alarm is raised. It should be clear that the detection rate is independent of user traffic (the persistence value of an atom does not depend on other atoms). The y-axis denotes an average number of false positives per day. Here, a false positive is simply a destination in the user traffic (assumed clean) which raised an alarm. The values are averaged across all users and over the last two weeks of user traffic data. This ROC curve is shown in Fig. 4. In an earlier section, we had selected a value for p^* based upon properties of the user generated whitelists; we can now determine the optimum value from the ROC curve. if we set $p^* \leq 0.6$, we are guaranteed to raise an alarm, for at least one of the atoms, in every botnet trace that we evaluated against. This brings the detection rate to 1.0. In general, lowering p^* acts to increase the detection rate (more botnet atoms cross the persistence bar) but also increases the false positive rate (more begign atoms also meet the standard). Selecting $p^* = 0.6$ seems balance the tension between these two points. At this operating point, we are able to detect every botnet and keep the false positives down to less than 1 a day on average. However, it is also important to study how this number varies across the user population.

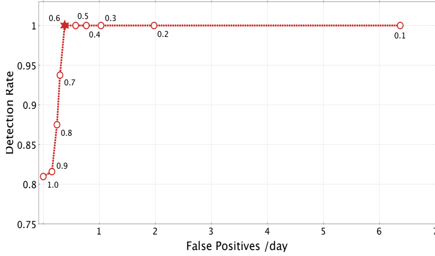


Fig. 4. RoC curve

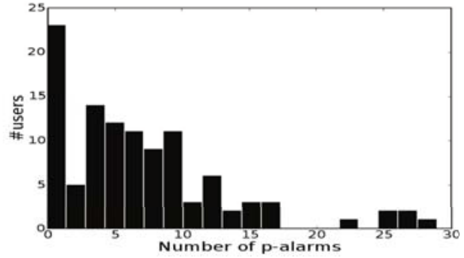


Fig. 5. Distribution of false positives

Figure 5 plots the histogram of false positives encountered by all the users, over the entire two week period, as determined by $p^* = 0.6$. A significant fraction of the population see few or no alarms in the two week period. A small handful of users—we speculate these are the very heavy traffic users—see 25-30 alarms over the entire period. To summarize the distribution, we see an average of 5.3 benign destination atoms being flagged as suspicious per user in the 2 week period. That is, the average user will have to dismiss fewer than 1 alarm every other day when this C&C detection system is in place on the users end-host. Since false positives are often associated with user annoyance, our method is able to carry out the detection with an extremely low user annoyance factor.

We note that the detection results presented here reflect the enterprise setting and may not generalize to other settings. P2P applications which (legitimately) connect to a large number of destinations might conceivably increase the false positives being experienced (we saw no p2p traffic in our dataset).

5.3 Detecting Botnet Attack Traffic

In the previous discussion, we focused on detecting C&C channels. Here we try to understand how our method can boost the detection rates of more traditional traffic feature (or volume) based anomaly detectors. Note that the whitelists constructed in the training phase can be considered the set of "known good destination" for a particular host. Thus, all traffic going to these destinations must be de-facto "anomaly free" and can be *filtered* out of the traffic stream being passed to a conventional anomaly detector.

The traditional anomaly detectors operate by tracking a time series of some significant traffic feature, e.g., number of outgoing connections in an interval, and raising an alarm when this crosses a threshold. The threshold is ideally determined based on the tail of the feature's distribution empirically derived from clean traffic. To distinguish the alarms in question from those triggered by C&C destinations (as discussed previously), we denote them "burst alarms". Thus, the persistence metric results in C&C alarms, and the anomaly detectors generate burst alarms. In the experimental evaluations we describe in the following, the (aggregate, without C&C filtered out) botnet traffic was superimposed on the traffic of each end-host. We point out that the botnet traces are generally shorter than the

user traces. To ensure that the trace overlay extends across the entire user trace, we replicated the botnet trace as often as necessary to fill up the entire testing window of 2 weeks.

There are a number of possible traffic features one can track, and a larger universe of anomaly detectors that can be defined on them. In the current instance, we use a simple connection count detector with a 99.9%-ile threshold. That is, the traffic feature of interest is the number of outgoing connections in 1 minute intervals, and the threshold is computed as the 99.9 percentile value of this distribution empirically computed from the training data. Specifically, we compare the detection results across two traffic streams, one where the outgoing traffic is filtered by the whitelist and the other where it is not. By "filter" we simply mean that all traffic to destinations on the whitelist is ignored and not passed along to the anomaly detector. Note that the same *definitional* threshold is used, i.e., the 99.9%-ile, but the values are different since the time-series are different (one of them has whitelisted destinations filtered out).

Figure 6 plots the detection rate over the entire user population. The x-axis enumerates the botnets from Table 2 and the y-axis is the fraction of users that generated a burst alarm for the particular botnet. The two bars correspond to the non-filtered (left column) and filtered (right column) traffic streams, the latter corresponding to our detection method. In the figure, we see a detection rate of 1.0 for some of the botnets, indicating *every* user generated an alarm when fed the tainted trace that included the traffic of (Gobot.T, AimBot-5, SpyBot-50, `storm`/Peed-69). In these cases, the filtering provides no added benefit. This is because the traffic volumes associated with these instances so egregious and beyond the range of normal traffic that any reasonable detector would raise an alarm. However, there are lots of other instances in the figure where detection with the filtered traffic is significantly better. For instance, in the case of `VB-666`,

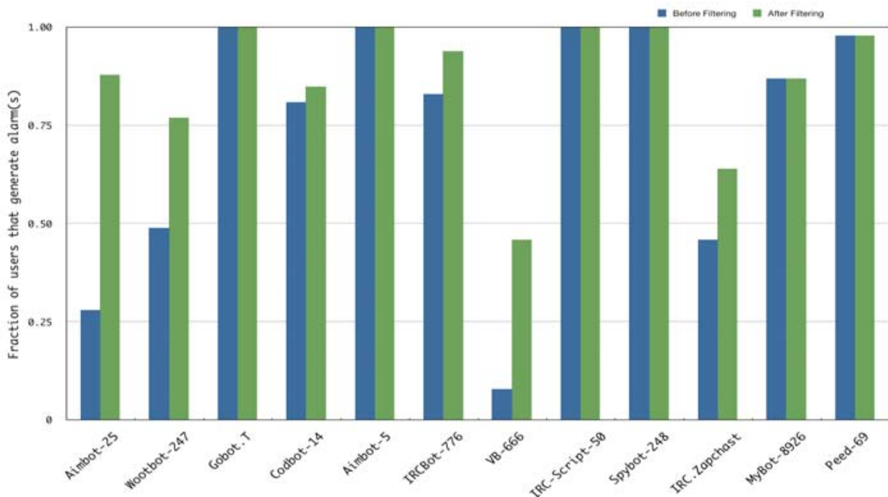


Fig. 6. Improvement in detection rate after filtering

which is the most dramatic result, we see a five fold improvement in detection when the traffic is filtered. Another example, with Aimbot-25, only 27% of the users generate an alarm in the general case, but this number grows to 85% when the traffic is filtered— a dramatic improvement. The intuition for why the filtering helps with the detection rate is thus: when the traffic to known good destinations is filtered out and the threshold recomputed, the new threshold tracks the residual traffic better and offers a small "gap" or range that is available for the botnet traffic. That is, as long as the volume of botnet traffic stays inside this gap it will fall under the threshold and be undetected. However this gap is small enough that even a small volume tends to go beyond the usable range. Clearly, the benefit of filtering the traffic is apparent when the botnet traffic volumes are low to moderate. When the volume is large, detection is easily carried out by any reasonably anomaly detector (even without filtering). Thus, filtering traffic through the whitelist helps to uncover stealthier traffic that is hidden inside. We carried out the same comparison for other detectors and found the results to be consistent with what we describe here. We omit details from these experiments for a lack of space.

Importantly, notice in the figure that for some of the botnet instances, the detection rate does not reach 100%, even with the filtering. This is possibly because of the variability in traffic across users: presumably, there is a sufficient gap between the traffic and the threshold for some users and the additional botnet traffic is able to squeeze into this gap. However, even when the volume based methods fail to carry out the detection to a complete degree, C&C alarms are generated for *every botnet trace* that we have collected and tested against (as shown in the previous discussion). Thus, even when the attack traffic is small enough to go undetected by the volume detectors, the botnets are still flagged by tracking persistence. This goes to underscore how critical it is to track temporal measures, rather than just volume, when dealing with botnets.

Thus, we demonstrate that by first learning whitelists of good destination atoms, and subsequently filtering out the traffic contributions of these destination atoms, we can dramatically improve the detection of botnet associated traffic. We enable more end-hosts to reliably generate alarms for this traffic, and to do so earlier.

6 Discussion

In this paper, we introduced the notion of "destination atoms" and "persistence" and subsequently described a method to detect the communication between a bot and its command and control infrastructure. The method we describe operates by associating a numeric value, quantified by persistence, with individual destination atoms. Destination atoms that are frequently contacted tend to have higher persistence values, even if the communication volume is low. Those with large persistence values are likely to be command and control servers. The method works best when the C&C infrastructure has a degree of centralization. We are also able to detect instances that are not centralized by exploiting

the regularity of communication to a small set of C&C destinations. However, botnets are constantly evolving with new cloaking and obfuscation mechanisms being developed constantly. For instance, Torpig [17] and Conficker [18] are recent botnets that make heavy use of domain name fluxing. Here, the botnets generate a very large number of domain names (typically based on a generating algorithm) to which connections are attempted. Such botnets are very hard to detect with general methods which do not try to target very specific botnet behaviors. Our method might not be successful at detecting these botnets since the names being generated, and to which connections are attempted, are completely unrelated and cannot be grouped into a smaller set of destination atoms. From the point of view of our detection method, these botnets seem to connect to a large number of destination atoms with uniformly low persistence values. However, methods have been developed in the recent past that specifically target fast flux networks [11]. This underscores the fact that a silver bullet approach to detecting and preventing botnets is unlikely to manifest anytime soon. A variety of methods focused on detecting different aspects of a botnets behavior are essential and must be used in conjunction to mitigate the botnet problem.

Another drawback to being a very general detection method is that the alarms generated are probabilistic in nature. The system, in of itself, cannot ascertain whether an alarm corresponds to an actual C&C destination or if it is benign. Processing an alarm must necessarily involve the end-user. If the end-user were to install a new application which begins to communicate with a particular server everytime it is launched, the behavior is likely to trigger an alarm at some point, which the user has to respond to. However, the act of installing a new application occurs infrequently and we don't believe the extra false positives due to this will be significant. Moreover, if our sytem were to be deployed inside an enterprise network, the alarms will be sent to a central console and processed by trained IT personnel; this would shift the onus away from end-users who may not be well equipped to decide if the alarm is benign.

Our method reliably detects all the malwares experimented with and does so with a very low false positive rate. The network traces used were collected in an enterprise network where p2p applications are prohibited (save for Skype, which serves a business purpose). The rate of false positives is likely to be much higher in the presence of significant p2p traffic. However, this is not really a bad result. Some of the botnets that p2p infrastructures for the command and control are fundamentally indistinguishable from actual p2p networks. In fact, the Storm botnet uses a legitimate (arguably) p2p network (Overnet) to host its command and control. One mechanism of dealing with this problem, where we want p2p like botnets to raise an alarm, but not legitimate p2p applications, would be to whitelist *applications* themselves. In this scenario, select legitimate applications would be whitelisted and all the traffic that they generate is considered de-facto legitimate; the whitelisted application would never trigger an alarm. Applications are almost always installed manually by the end-users which implicitly implies a trust relationship between the user and application.

7 Conclusions

In this paper, we introduced the notion of "persistence" as a temporal measure of regularity in connection to "destination atoms", which are destination aggregates. We then described a method that builds whitelists of known good destination atoms in order to isolate *persistent* destinations in the traffic which are likely C&C channels. The notion of persistence, a key contribution of this work, turns out to be critical in detecting the covert channel communication of botnets. Moreover, being a very coarse measure, persistence does not require any protocol semantics or to look inside payloads to detect the malware. Using a large corpus of (clean) user traffic as well as a collection of botnet traces, we showed that our method successfully identified C&C destinations in each and every botnet instance experimented with, even the ones that make very few connections. We demonstrated that this detection incurs low overhead and also has a low user annoyance factor. Even though our method is focused on uncovering C&C communication with botnets that have a centralized infrastructure, we are able to uncover even those that are not, as long as there is a certain regularity (even over short time scales) in communicating with the C&C destinations.

In the future, a key task that we would like to undertake is to run our method on a much larger sample of botnet traffic than we were able to collect on our own, and perhaps validate it against a longer trace of user traffic data. Unfortunately, as we learned in the course of this work, such an effort requires a significant amount of resources, and technical expertise. This goes to show that a much larger community effort is needed to collect, share and annotate traces to support research efforts in designing botnet mitigation solutions in particular, and security mechanisms in general.

References

1. de Oliveira, K.C.: Botconomics: Mastering the Underground Economy of Botnets. FIRST Technical Colloquium
2. McAfee Corp.: Avert Labs Threat Predictions for 2009, `http://www.mcafee.com/us/local_content/reports/2009_threat_predictions_report.pdf`
3. Cooke, E., Jahanian, F., McPherson, D.: The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In: Proceedings of the Workshop on Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI 2005), Berkeley, CA, USA, p. 6. USENIX Association (2005)
4. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow Anomaly Detection. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 48–62. IEEE Computer Society, Los Alamitos (2006)
5. Gao, D., Reiter, M.K., Song, D.: On Gray-box Program Tracking for Anomaly Detection. In: Proceedings of the 13th USENIX Security Symposium, Berkeley, CA, USA, p. 8. USENIX Association (2004)
6. Binkley, J.R., Singh, S.: An Algorithm for Anomaly-based Botnet Detection. In: Proceedings of the 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI 2006), Berkeley, CA, USA, p. 7. USENIX Association (2006)

7. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection through IDS-Driven Dialog Correlation. In: Proceedings of 16th USENIX Security Symposium, Berkeley, CA, USA, pp. 1–16. USENIX Association (2007)
8. Gu, G., Zhang, J., Lee, W.: BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In: Proceedings of the Annual Network and Distributed System Security Symposium (NDSS 2008) (Febuary 2008)
9. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-independent Botnet Detection. In: Proceedings of the 17th Usenix Security Symposium, Berkeley, CA, USA, pp. 139–154. USENIX Association (2008)
10. Kreibich, C., Kanich, C., Levchenko, K., Enright, B., Voelker, G., Paxson, V., Savage, S.: On the Spam Campaign Trail. In: First USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET 2008 (2008)
11. Holz, T., Gorecki, C., Rieck, K., Freiling, F.: Measuring and Detecting Fast-Flux Service Networks. In: Proceedings of the Annual Network and Distributed System Security Symposium, NDSS 2008 (2008)
12. Jung, J., Paxson, V., Berger, A., Balakrishnan, H.: Fast Portscan Detection using Sequential Hypothesis Testing. In: IEEE Symposium on Security and Privacy, pp. 211–225 (2004)
13. Sekar, V., Xie, Y., Reiter, M.K., Zhang, H.: Is Host-Based Anomaly Detection + Temporal Correlation = Worm Causality? Technical Report CMU-CS-07-112, Carnegie Mellon University (March 2007)
14. McDaniel, P.D., Sen, S., Spatscheck, O., van der Merwe, J.E., Aiello, W., Kalmanek, C.R.: Enterprise Security: A Community of Interest Based Approach. In: Proceedings of the Annual Network and Distributed System Security Symposium, NDSS 2006 (2006)
15. ClamAV: Clam AntiVirus, `http://www.clamav.net`
16. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks (1999)
17. Stone-Gross, R., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., Vigna, G.: Your Botnet is My Botnet: Analysis of a Botnet Takeover (May 2009), `http://www.cs.ucsb.edu/~seclab/projects/torpig/torpig.pdf`
18. Porras, P., Saidi, H., Yegneswaran, V.: An Analysis of Conficker's Logic and Rendezvous Points. Technical report, SRI International (March 2009)

An Experimental Study on Instance Selection Schemes for Efficient Network Anomaly Detection

Yang Li[1,2], Li Guo[2], Bin-Xing Fang[2], Xiang-Tao Liu[2], and Lin-Qi[3]

[1] China Mobile Research Institute, Beijing, China 100053
[2] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China 100190
[3] University of Maryland, America 21226
samsunglinux@163.com

Abstract. Traditional researches on network anomaly detection have been solely focused on the detection algorithms, whereas an important issue that has not been well studied so far is the selection of normal training data for network anomaly detection algorithm, which is highly related to the detection performance and computational complexities. In this poster, we present two instance selection mechanism – *EFCM (Enhanced Fuzzy C-Means)* as well as *GA (Genetic Algorithm)* for network anomaly detection algorithm, aiming at limiting the size of training dataset, thus reducing the computational cost of them, as well as boosting their detection performance. We report our experimental results on several classic network anomaly detection algorithms by using the network traffic trace collected from a real network environment.

To our best knowledge, this topic about instance selection for efficient network anomaly detection is still hardly well addressed in any published literatures. Most of current network anomaly detection algorithms are highly dependent on the training dataset [1, 2], if the dataset is of poor quality or its size is too big, the detection performance would degrade greatly. The large size dataset would also cause additional expensive computational costs. In this poster, we therefore propose two instance selection schemes for anomaly detection algorithms: FCM-based instance selection mechanism – EFCM (Enhanced Fuzzy C-Means) and Genetic Algorithm (GA), and provide the preliminary results.

As for ***EFCM algorithm***, we first utilize it to cluster the original training dataset into three classes: *notable data*, *obscure data* and *redundant data*. We refer to the notable data as what belong to and represent a series of clusters; the obscure data denotes those that belong to any cluster with very small (or under a very small constant) membership grades calculated by FCM; the redundant data then includes the remaining data that don't belong to any of the above two classes. We then select some high quality "representatives" as our training subset, which contains the most useful data that contribute the currently existing anomaly detection techniques to distinguish the normal from the abnormal traffic. ***Genetic Algorithm*** (GA) is optimization algorithm based on natural genetics and selection mechanisms. We utilize it to fulfill the instance selection task. Training dataset is denoted as *TR* with instances, and the search space associated with the instance selection of *TR* is constituted by all the subsets of *TR*. Then, the chromosomes should represent subsets of *TR*. This is accomplished by using a binary representation. A chromosome consists of genes (one for each instance in *TR*) with two possible states: 0 and 1. If the gene is 1, then its associated instance is included in the subset of

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 346–347, 2009.
© Springer-Verlag Berlin Heidelberg 2009

TR represented by the chromosome. If it is 0, then this instance does not occur. After running GA algorithm, the selected chromosomes would be the reduced training dataset for network anomaly detection algorithms.

From our preliminary experimental studies on EFCM and GA (see Table 1 and Table 2) over real network traffic trace, we found many useful results: Firstly, after employing instance selection mechanisms, we found their computational costs reduced greatly, whereas their detection performances still held high or became even better, which is an very important finding. For example, as for TCM-KNN algorithm [2], the size of training dataset is reduced almost 90%, and the training and detection time is reduced about 77%. Secondly, the detection rate (TP) appears a little decrease, while the false positive rate (FP) demonstrates an inspiring trend of decrease. Thirdly, the effectiveness of EFCM and GA methods in network anomaly detection domain are comparable. Therefore, from these results we are apt to select EFCM as instance selection methods for network anomaly detection algorithms in real network environment, while under the circumstances of having no restrict requirements on the training speed, GA may be more suitable for leading to better detection performance.

Table 1. Comparison Results after Using Instance Selection for Various Algorithms

Algorithm	Without instance selection		EFCM		GA	
	TP	FP	TP	FP	TP	FP
One-classSVM[1]	95.85%	8.67%	94.98%	9.87%	95.72%	7.03%
Fixed-width Clustering[1]	72.57%	15.37%	72.77%	13.48%	72.55%	10.26%
KNN score[1]	92.72%	10.63%	88.78%	9.03%	92.08%	8.78%
TCM-KNN[2]	100%	1.29%	99.38%	1.07%	99.46%	0.98%

Table 2. Computational Costs after Instance Selection for Various Algorithms

Algorithm	Without instance selection		EFCM		GA	
	Training time	Detection time	Training time	Detection time	Training time	Detection time
One-class SVM	/	26200s	/	8680s	/	7287s
Fixed-width Clustering	/	1098s	/	213s	/	210s
KNN score	/	29898s	/	1398s	/	1293s
TCM-KNN	40418s	0.4558s	992s	0.1007s	983s	0.0987s

References

1. Eskin, E., Arnold, A., Prerau, M., Portnoy, L., Stolfo, S.: A geometric framework for unsupervised anomaly detection: detecting intrusions in unlabeled data. In: Proc. ADMCS 2002, pp. 78–99 (2002)
2. Li, Y., Fang, B.X., Guo, L., Chen, Y.: Network Anomaly Detection Based on TCM-KNN Algorithm. In: Proc. ACM ASIACCS 2007, pp. 13–19 (2007)

Automatic Software Instrumentation for the Detection of Non-control-data Attacks

Jonathan-Christofer Demay, Éric Totel, and Frédéric Tronel

SUPELEC, Rennes, France
{first_name.last_name}@supelec.fr

Abstract. To detect intrusions resulting of an attack that corrupted data items used by a program to perform its computation, we propose an approach that automatically instruments programs to control a data-based behavior model during their execution. We build our model by discovering the sets of data the system calls depend on and which constraints these sets must verify at runtime. We have implemented our approach using a static analysis framework called *Frama-C* and we present the results of experimentations on a vulnerable version of *OpenSSH*.

To make a program deviate from its specification, an intrusion needs to corrupt some data initially used by the process to control its execution (control-data) or to perform its computation (non-control-data). To execute illegal system calls, an attack can use an invalid code (injected or out-of-context) by corrupting the first class of data or use a valid code (with invalid inputs or through an invalid path) by corrupting the second class of data. To detect intrusions, anomaly-based intrusion detection systems check for deviations from a model reflecting the normal behavior of programs. Numerous of them working at the system level build their model using sequences of system calls. This approach detects various control-data attacks but misses most of the non-control-data ones. Furthermore, evasion techniques such as mimicry attacks can be used to bypass these detection mechanisms during a control-data attack. Several enhancements of this approach have been proposed, notably by adding information available at the system level, such as the parameters of the system calls or their execution context. The detection of control-data attacks is improved both in accuracy and completeness, but non-control-data attacks remain mostly undetected [1].

Our work focuses on the detection of non-control-data attacks by checking for memory corruptions that may lead to illegal system calls executed by a valid code. Since those attacks need to corrupt specific data items with specific values, they may put the memory of the process in an inconsistent state regarding the specification of the program. Our approach consists in finding consistency properties in a program through static analysis to detect data corruptions induced by non-control-data attacks. We thus build a data-oriented model based on these properties to detect such attacks inside the program at runtime. To derive these properties from the source code, two problems need to be addressed : for a particular system call SC_i, what is the set of variables V_i that influence

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 348–349, 2009.
© Springer-Verlag Berlin Heidelberg 2009

its execution, and what is the set of constraints C_i these variables must verify. Thus, we define for a given system call its normal data behavior by the triple (SC_i, V_i, C_i). We can then define our data behavior model by $\{\forall i, (SC_i, V_i, C_i)\}$.

To build the set of intrusion sensitive variables, we must consider two kinds of dependency relations : value dependencies, that influence the parameters of system calls, and control dependencies, that influence the paths that lead to them. Discovering these relations through static analysis can be done using program slicing techniques. A program slice can be defined as the subset of instructions that influence the execution of a particular instruction from this program called a point of interest. The set of variables V_i on which a system call SC_i depends is the set of variables used by the program slice computed at the point of interest SC_i. To discover constraints that an attack could broke, we must consider the dependency relations that may exists between them : that is why the variables from a set should be processed as a tuple and not individually. We choose to limit the constraints C_i a given set V_i must verify to the variation domain of the corresponding tuple. Computing these variation domains through static analysis can be done using abstract interpretation techniques, which produce an over approximation of the semantic of a program based on lattices. To detect deviations from our data behavior model, we choose to insert reasonableness checks in the program that will raise an alert when a constraint is broken. To derive such checks from our model, we have a reachability problem to address : all the variables V_i a system call SC_i depends on may not be reachable at the calling point in the source code. We choose to solve this by distributing the verification of C_i along the call stack that lead to SC_i at each function call, checking only the subset of V_i containing all the variables reachable at each point.

To evaluate our approach against non-control-data attacks on real vulnerabilities [1], we have developed a tool that implemented our detection model using *Frama-C* [2], a modular framework dedicated to the analysis of source codes of programs written in *C*. Our tool is a source to source translator that transforms an untrusted program into a data security enforced program. It computes a normal data behavior model from the source code and inserts in the program executable assertions derived from this model. We have tested our tool on a vulnerable version *OpenSSH* which suffer from an integer overflow allowing remote attackers to write arbitrary values to any location in the memory of the process. After using our tool on its source code, 8% of the function calls are checked for inconsistencies, which detect the two known non-control-data attacks exploiting this vulnerability, with an overhead of 0.5%. This shows that our approach can indeed detect non-control-data attacks at runtime while inducing a low overhead.

References

1. Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer: Non-control-data Attacks are Realistic Threats. Usenix Security Symposium, 2005
2. *Frama-C*: Framework for Modular Analysis of C. http://frama-c.cea.fr/index.html

BLADE: Slashing the Invisible Channel of Drive-by Download Malware

Long Lu[1], Vinod Yegneswaran[2], Phillip Porras[2], and Wenke Lee[1]

[1] School of Computer Science, Georgia Tech, Atlanta, GA 30332 USA
[2] SRI International, Menlo Park, CA, 94025 USA
{long,wenke}@cc.gatech.edu, {vinod,porras}@csl.sri.com

Abstract. Drive-by downloads, which result in the unauthorized installation of code through the browser and into the victim host, have become one of the dominant means through which mass infections now occur. We present BLADE (Block All Drive-by download Exploits), a browser-independent system that seeks to eliminate the drive-by threat. BLADE prudently assumes that the legitimate download of any executable must result from explicit user consent. BLADE transparently redirects every browser download into a non-executable *safe zone* on disk, unless it is associated with a programmatically inferred *user-consent event*. BLADE thwarts the necessary underlying transaction on which all drive-by downloads rely, therefore it requires no prior knowledge of the exploit methods, and is not subject to circumvention by obfuscations or zero-day threats.

1 The BLADE System

Unlike push-based approaches adopted by Internet scanning worms and viruses, contemporary malware publishers rely on drive-by exploits for silent dissemination of spyware, trojans, and bots [2]. As a countermeasure, BLADE is a kernel-based monitor designed to block all malware delivered without user knowledge via browsers and overcomes the challenges described in [1].

BLADE's design is motivated by the fundamental observation that all browser downloads fall into either of two basic categories: supported file types (e.g., html, jpeg) or unsupported file types (e.g., exe, zip). While browsers silently fetch and render all supported file types, they must prompt the user when an unsupported-type is encountered. The objective of client-side download exploits is to deliver malicious (*unsupported*) content through the browser using methods that essentially bypass the standard unsupported-type user prompt interactions. BLADE's approach is to intercept and impose "execution prevention" of all downloaded content that has not been directly consented to by user-to-browser interaction. To achieve this, BLADE introduces two key OS-level capabilities:

(1) User-Interaction Tracking. A novel aspect of BLADE is the introduction of user-interaction tracking as a means to discern *transparent* browser downloads from those that involve direct user authorization. Operating from the kernel space, BLADE introduces a browser-independent *supervisor*, which infers user

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 350–352, 2009.
© Springer-Verlag Berlin Heidelberg 2009

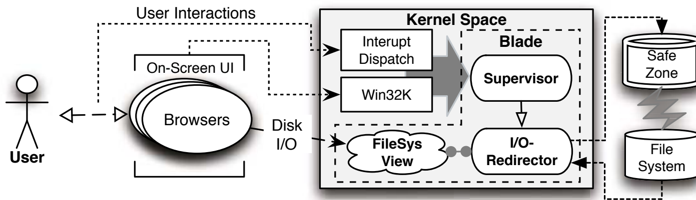


Fig. 1. BLADE's Architecture

consent-to-download events, by reconstructing on-screen user interfaces (UI) from kernel memory and monitoring user interactions in the form of hardware interrupts. Specifically, it retrieves semantic UI information from the kernel objects maintained by the windowing subsystem (Win32K), discovers interested UI elements and their status changes (e.g., download confirmation dialogs), and listens to hardware-interaction events (e.g., mouse clicks) targeted at any interested UI element. Once a download consent event is inferred, the supervisor records it as an authorization along with the information parsed from UI elements (e.g., file names and URLs).

(2) Disk I/O Redirection. BLADE's *I/O-Redirector* transparently redirects all hard disk write operations to a *safe zone*. This safe zone, created and managed by BLADE, represents offshore storage inaccessible from the local file system. Being addressable only through BLADE ensures that files in the safe zone can never be loaded or executed, even by the OS. Upon finishing each file write operation, the *I/O-Redirector* queries the *supervisor* and maps the current file to the local file system if a stored authorization correlates with it. To maintain functional consistency, the supervised processes are also provided a modified file system view, which renders the impression that all disk writes are carried out in their respective locations, while actual disk I/O to these files are forwarded by BLADE to the safe zone. A prototype of BLADE is now under development as a kernel driver for Windows platforms, which will be tested with multiple versions of Firefox, Internet Explorer and Chrome.

Threat Model. We assume that the OS, the underlying hardware and network infrastructure are uncompromised. The attacker's goal is to perform a forced upload and execution of malicious binary content on the victim machine. Upon successfully hijacking control of a browser process, an attacker may pursue either of two possible paths to bypass BLADE and install a malware binary: (a) evading I/O redirection, or (b) executing the malware stored in the safe zone. As a kernel driver only dealing with trusted OS components and unforgeable hardware events (e.g., mouse clicks), BLADE is not subject to code injection or data manipulation attacks, and not deceived by fake UI messages which makes (a) difficult. Likewise, attempts to launch the malware from outside the browser process are naturally prevented as the the malware is only addressable through BLADE.

References

1. Egele, M., Kirda, E., Kruegel, C.: Mitigating drive-by download attacks: Challenges and open problems. In: iNetSec 2009, Zurich, Switzerland (April 2009)
2. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iframes point to us. In: Proceedings of the 17th USENIX Security Symposium (2008)

CERN Investigation of Network Behaviour and Anomaly Detection

Milosz Marian Hulboj and Ryszard Erazm Jurga

CERN — HP Procurve openlab project
CH-1211, Geneve 23, Switzerland
{mhulboj,rjurga}@cern.ch
http://cern.ch/openlab

Abstract. The CINBAD (CERN Investigation of Network Behaviour and Anomaly Detection) project was launched in 2007 in collaboration with ProCurve Networking by HP. The project mission is to understand the behaviour of large computer networks in the context of high performance computing and large campus installations such as at CERN, whose network today counts roughly 70,000 Gigabit user ports. The goals of the project are to be able to detect traffic anomalies in such systems, perform trend analysis, automatically take counter measures and provide post-mortem analysis facilities. This paper will present the main project principles, data sources, data collection and analysis approaches as well as the initial findings.

Keywords: computer networks, anomaly detection, packet sampling, network monitoring.

1 Network Anomalies

Anomalies are nowadays a fact of life in computer networks. However anomaly definition is very domain specific and the causes are diverse (network faults, malicious attacks, viruses and worms, misconfiguration, policy violations, etc).

The following common denominator can be factored out: an anomaly is always a deviation of the system from the normal (expected) behaviour (baseline); the normal behaviour (baseline) is never stationary and anomalies are not always easy to define. As a consequence, non-trivial anomalies are not easy to detect.

2 sFlow Packet Sampling and Other Data Sources

With the modern high-speed networks it is impossible to monitor all the packets traversing the links. sFlow, the industry standard for monitoring high-speed switched networks overcomes this issue by providing randomly sampled packets (first 128 bytes) from the network traffic. These initial bytes provide crucial information for the analysis conducted by the CINBAD team. Our collection and analysis is based on traffic from more than 1500 switches and routers around

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 353–354, 2009.
© Springer-Verlag Berlin Heidelberg 2009

CERN. The CINBAD team also investigates other data sources that can be used to augment the information provided by the packet sampling. At CERN we can use the reports from the central antivirus service, detailed logs from the DNS servers and other central services. Information from many different data sources may be correlated in order to find interesting phenomena.

3 Data Analysis

We have been investigating various data analysis approaches that could be categorised mainly into the two domains: statistical and signature based analysis. The former depends on detecting deviations from normal network behaviour while the latter uses existing problem signatures and matches them against the current state of the network. The signature based approach has numerous practical applications with SNORT (an opensource intrusion detection system) being a prominent example. The CINBAD team has successfully ported SNORT and adapted various rules in order to work with sampled data. It seems to perform well, and provides a low false positive rate. However, the system is blind and can yield false negatives in case of unknown anomalies. Fortunately, this problem can be addressed by the statistical approach. This requires the understanding of the normal network behaviour. Expected network activity can be established by specifying the allowed patterns in certain parts of the network. While this method can work very well for a DNS or Web server that are supposed to be contacted only on a given protocol port number, for more general purposes this approach would not scale.

A second approach to infer the normal network behaviour is to build various network profiles by learning from the past. Selection of robust metrics that are resistant to data randomness plays important role in characterising the expected network behaviour. Once these normal profiles are well established the statistical approach can detect new and unknown anomalies. However, this might not provide sufficient information to identify the anomaly type. The CINBAD project combines the statistical approach with the signature based analysis in order to benefit from the synergy of the two techniques. While the latter provides the detection system with a fast and reliable detection rate, the former is used to detect the unknown anomalies and to produce new signatures.

4 Initial Findings

By using the described approach the CINBAD team performs constant monitoring of both campus and Internet traffic. A certain number of misbehaviours were identified, for example: DNS abuse, p2p applications, rogue DHCP servers, worms, trojans, unauthorised wireless base stations, etc. Some of our findings resulted in the refinement of the security policies.

Reference

1. Jurga, R., Hulboj, M.: Technical Report Packet Sampling for Network Monitoring

Blare Tools: A Policy-Based Intrusion Detection System Automatically Set by the Security Policy

Laurent George, Valérie Viet Triem Tong, and Ludovic Mé

SUPELEC, SSIR Group (EA 4039), Rennes, France
firstname.lastname@supelec.fr

Abstract. We present here an intrusion detection system automatically parameterized by the security policy. The main idea consists in monitoring information flows in an operating system in order to detect those not allowed by the security policy. In previous works ([1,2] and [3]), the security policy set at the initialization of the IDS and can not be updated. We focus here on the dynamism of the security policy monitored.

A security policy defines the *authorized behavior* of users and applications in a system, generally in terms of access rights (read/write). We use such a policy to infer a specification of the *reference behavior* for our IDS. The access control policy is translated into an information flow policy in which a piece of information i is allowed to flow into an information container c if there exists at least one user allowed to read i and to write into c. Pieces of information (or atomic information) in the system are characterized by the name of their original container and the information flow policy is a relation between atomic information and containers of information. Once the information flow policy $\mathcal{P}$ is defined, containers of information are tagged with three *tags* T^I, T^R and T^W. For a given container of information c, the first tag T^I lists atomic information that are origins of the current content of c (1). T^R is the set of elements in $\mathcal{P}$ related to information listed in T^I (2), the last tag T^W lists all elements in $\mathcal{P}$ related to c (3). At each observation of an information flow, tags T^I and T^R are updated to keep properties (1) and (2), the last tag never changes. An object with disjoints tags T^R and T^W is a witness of an illegal information flow and generates an alert. Indeed, in this case, the pair (information, container) is not in the defined flow policy. Consider now that the security policy is modified, for instance a user u gaining a read access to an object o. In terms of information flows two interpretations are possible: u may now legally access to the original contains of o or u may now legally access to information currently contained by o. The two interpretations can be expressed as new element(s) added to $\mathcal{P}$. The IDS has to be updated since u can now legally access new atomic information wherever these information are located. For that purpose, tags of relevant objects are updated. We have formally defined an update mechanism and proved that properties (1) and (2) still hold.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 355–356, 2009.
© Springer-Verlag Berlin Heidelberg 2009

In our current implementation, objects are all files, sockets and memory files of an operating system. Our IDS is composed of two consoles and a policy controller. The read and write tags are implemented with a binary vector linked to each object. The information tag is implemented with a sorted list of integers, each integer being associated to an atomic information. The first console, Blare, observes information flows between objects at the system level. Blare is a patch for standard Linux kernel where any system call that engender information flows call our functions `flow_in` and `flow_out`. This two functions update the tags of concerned objects and computes intersection of the tags T^R and T^W. An alert is raised when empty. Blare provides tools in userspace: `lsinfo` (resp. `setinfo`) to list (resp. to set) T^I and `findinfo` to find all the objects of the system containing a subset of a list of atomic information. The second console JBlare is a patch for the Java Virtual Machine and refines the view of Blare by observing information flow through method calls in Java programs. JBlare is responsible of tags T^R for objects output by a java program. The policy controller waits for any access control changes. It translates these changes in terms of information flow policy and uses `findinfo` and `setref` to update the relevant tags. The administrator uses the controller to declassify information if needed. In this case the controller calls `setinfo` to rename informations and `setref` to attribute new rights to these information.

With the Blare project and the Blare tools, we aim at proposing a set of collaborative tools able to monitor information flows at several level of granularity. Beside this practical objective, we also aims at building this tools upon a formal intrusion detection model allowing to prove the completeness and soundness (for information flow observable at a given level) of the detection. We now envision other tools to complete the set. For example, we aim at applying our model for the monitoring of web services. Such services typically consist of an operating system, a programming language runtime system (e.g. the JVM), an application framework and the application software itself. We already have applied our approach to the first two components. Now, we aims at studying its application to the application framework. However, Since Web-services- based applications are by nature distributed systems, it is necessary to analyse potential threats as multi-step attack scenarios across multiple nodes and to adapt the model and its implementation to such a context.

References

1. Zimmermann, J., Mé, L., Bidan, C.: Introducing reference flow control for detecting intrusion symptoms at the OS level. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, p. 292. Springer, Heidelberg (2002)
2. Zimmermann, J., Mé, L., Bidan, C.: An improved reference flow control model for policy-based intrusion detection. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 291–308. Springer, Heidelberg (2003)
3. Hiet, G., Viet Triem Tong, V., Mé, L., Morin, B.: Policy-based intrusion detection in web applications by monitoring java information flows. In: 3nd International Conference on Risks and Security of Internet and Systems (2008)

Detection, Alert and Response to Malicious Behavior in Mobile Devices: Knowledge-Based Approach

Asaf Shabtai, Uri Kanonov, and Yuval Elovici

Deutsche Telekom Laboratories at Ben-Gurion University and
Department of Information Systems Engineering, Ben-Gurion University, Israel
{shabtaia,kanonov,elovici}@bgu.ac.il

Abstract. In this research, we evaluate a knowledge-based approach for detecting instances of known classes of mobile devices malware based on their temporal behavior. The framework relies on lightweight agent that continuously monitors time-stamped security data within the mobile device and then processes the data using a light version of the Knowledge-Based Temporal Abstraction (KBTA) methodology. The new approach was applied for detecting malware on Google Android powered-devices. Evaluation results demonstrated the effectiveness of the proposed approach.

Keywords: KBTA, Host-Based Intrusion Detection Systems, Mobile Devices.

Smartphones have evolved from simple mobile phones into sophisticated yet compact minicomputers. Mobile devices become susceptible to various new threats such as viruses, Trojans and worms, all of which are well-known from the desktop computers. In this research we examine the applicability of detecting malware instances using a light version of the KBTA method [1] that can be activated on resource-limited devices. Using KBTA, continuously measured data (e.g., number of running processes) and events (e.g., software installation) are integrated with a temporal-abstraction knowledge-base; i.e., a security ontology for abstracting higher-level, meaningful concepts and patterns from raw, time-oriented security data, also known as temporal abstractions (Fig. 1). Automatically-generated temporal abstractions can be monitored to detect suspicious temporal patterns and to invoke the proper response. These patterns are compatible with a set of predefined classes of malware as defined by a security expert employing a set of time and value constraints. Derived patterns are also context-aware, thus the same data may be interpreted differently within different contexts.

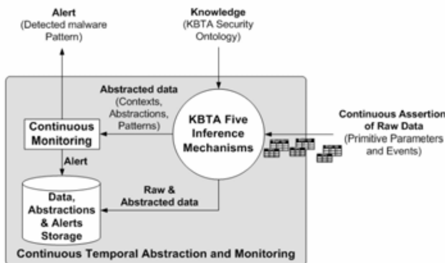


Fig. 1. The KBTA process

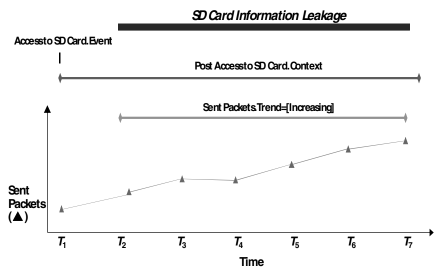


Fig. 2. SD-Card information leakage pattern

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 357–358, 2009.
© Springer-Verlag Berlin Heidelberg 2009

For example, the “SD-card Information Leakage” pattern is derived when an “Access to SD-card” event is detected and generates the “Post Access to SD-card” Context. Then, within the “Post Access to SD-card” context, an increasing number of “Sent Packets” (Trend) is derived which may be a result of a malicious software sending the content stored on the SD-card to a remote server (Fig. 2). Following is the pattern definition in CAPSUL language [2]:

Linear Pattern: SD-card Information Leakage pattern
  Context: Post Access to SD-Card
  Linear Components:
    Parameter Component:
      Sent Packets Trend
      Abstracted From:
        Sent Packets
      Local Constraints:
        value = INCREASING
        duration > 3sec
  Output Value of Pattern:
    Value Function: value = SD-Card Information Leakage

For our evaluation, we used a host-based intrusion detection system (HIDS) that was developed for Google Android powered-devices and which monitors more than 100 raw parameters and events, including some that were used to define basic abstractions and complex patterns. The HIDS employs a light-weight Incremental KBTA detector. A smartphone security ontology for detecting the malware was defined. The ontology includes five different temporal patterns of potentially malicious behavior: Denial of Service (overloading the system CPU and memory), abuse of SMS messages, abuse of the device camera, injecting malware via USB connection and theft of confidential information stored on the SD-Card. We have developed five different smartphone malware and later deliberately infected a G1 Android device. Evaluation results showed the ability to detect the malware while reducing false alarms using context-based interpretations. When an alert is detected, in addition to displaying a notification, the Android HIDS is capable of employing effective countermeasures tailored for the alert (e.g., all network transports can be disconnected when information leakage is detected or the malicious application can be terminated).

The HIDS can quickly be adapted to new malware classes by simply modifying the knowledge-base. In addition, KBTA defines patterns in a fuzzy fashion as a set of constraints, rather than as a hard-coded signature for each and every known malware. Consequently, it facilitates detection of instances of malware even when they have not been encountered before. The system can also help in integrating alerts from other sensors as primitive parameters. Meshing alerts from multiple sensors reduces the amounts of false alarms and isolates solid and reliable alerts, such that persist for a substantial time interval.

References

1. Shabtai, A., Fledel, Y., Elovici, Y., Shahar, Y.: Using the KBTA Method for Inferring Computer and Network Security Alerts from Time-stamped, Raw System Metrics. Journal in Computer Virology (2009), doi:10.1007/s11416-009-0125-5
2. Chakravarty, S., Shahar, Y.: CAPSUL - A constraint-based specification of repeating patterns in time-oriented data. Annals of Mathematics and AI 30(1-4), 3–22 (2000)

Autonomic Intrusion Detection System

Wei Wang[1,⋆], Thomas Guyet[2,3], and Svein J. Knapskog[1]

[1] Q2S Centre, Norwegian University of Science and Technology (NTNU)
wwangemail@gmail.com
[2] Project DREAM, INRIA Rennes/IRISA, France
[3] AGROCAMPUS OUEST, Rennes, France

Abstract. We propose a novel framework of autonomic intrusion detection that fulfills online and adaptive intrusion detection in unlabeled audit data streams. The framework owns ability of self-managing: self-labeling, self-updating and self-adapting. Affinity Propagation (AP) uses the framework to learn a subject's behavior through dynamical clustering of the streaming data. The testing results with a large real HTTP log stream demonstrate the effectiveness and efficiency of the method.

1 Problem Statement, Motivation and Solution

Anomaly Intrusion Detection Systems (IDS) are important in current network security framework. Insomuch as data involved in current network environments evolves continuously and as the normal behavior of a subject may have some changes over time, a static anomaly IDS is often ineffective. The detection models should be frequently updated by incorporating new incoming normal examples and be adapted to behavioral changes. To achieve this goal, there are at least two main difficulties: (1) the lack of precisely labeled data that is very difficult to obtain in practice; (2) the streaming nature of the data with behavioral changes.

To tackle these difficulties, we propose a framework of autonomic IDS that works in a fashion of self-managing, adapting to unpredictable changes whilst hiding intrinsic complexity to operators. It has abilities of self-labeling, self-updating and self-adapting for detecting attacks over unlabeled data streams.

The framework is under **an assumption of rareness of abnormal data**. We thus "capture" the anomalies by finding outliers in the data streams. Given a bunch of data stream, our method identifies outliers through the initial clustering. In the framework, the **detection model** is a set of clusters of normal data items. The outliers generated during the clustering as well as any incoming outlier that is too far from the current model are suspected to be attacks. To refine our diagnosis, we define three states of a data item: *normal*, *suspicious* and *anomalous*. If an outlier is identified, it is marked as *suspicious* and then put

⋆ The work was supported by Q2S Centre in communication systems, Centre of Excellence, which is appointed by the Research Council of Norway and funded by the Research Council, NTNU and UNINETT. The work was also supported by ERCIM fellowship program. The authors thank Florent Masseglia for providing us the data.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 359–361, 2009.
© Springer-Verlag Berlin Heidelberg 2009

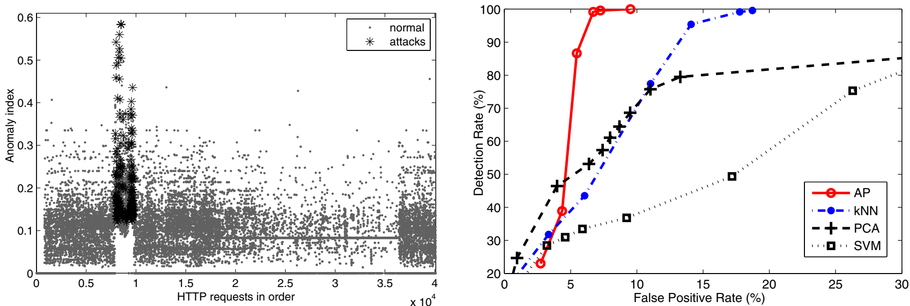
into a **reservoir**. Otherwise, the detection model is **updated** with the normal incoming data until a change is found, triggering model rebuilding to **adapt** to the current behavior. A suspicious item is considered as real *anomalous* if it is again marked as suspicious after the adaption.

2 Implementation and Discussion

The autonomic IDS is effective for detecting rare attacks [1]. Detecting bursty attacks is a challenge as the attack scenario does not well match the assumption. We thus design another two mechanisms during the autonomic detection. First, if a data item is very far from the model, the data item will be flagged as *anomalous* immediately (other than considered as *suspicious*). Second, a change is triggered if the percentage of outliers is high (e.g., larger than 60%) during a time period. Bursty attacks can thus be easily identified by the large dissimilarity and by the prompt model rebuilding.

We use Affinity Propagation (AP) and StrAP [2] to detect bursty attacks with the framework. We use a real HTTP log stream to test the method. Character distribution of each HTTP request is used as the feature and the IDS is to identify whether a request is normal or not. The data contains 40,095 requests in which 239 attacks occurring in a very short interval (request 7923-9743th, see Fig.1(a), the k-NN distance between a data item and the training items) after filtering out static requests. To facilitate comparison, we also use another three static methods k-NN, PCA and one class SVM for the detection. The first 800 *attack-free* requests are used for training the static models while the first 800 requests are used for AP initial clustering. Testing results are shown in Fig.1(b).

Fig.1(a) shows that the normal behavior changes over time and Fig.1(b) indicates that the autonomic detection method achieves the better results than other three static methods while the detection rates are higher than 50%. Note that the autonomic IDS does not need *a priori* knowledge while static methods need labeled data for training. Our future work is combining the autonomic IDS with effective static methods to prevent mimicry attacks (e.g., implementing large-scale attacks to evade the autonomic IDS).



(a) Distance distribution of the log stream (b) Testing results with comparison

Fig. 1. Dynamic normal behaviors and testing results with comparison

References

1. Wang, W., Masseglia, F., Guyet, T., Quiniou, R., Cordier, M.O.: A general framework for adaptive and online detection of web attacks. In: WWW, pp. 1141–1142 (2009)
2. Zhang, X., Furtlehner, C., Sebag, M.: Data streaming with affinity propagation. In: Daelemans, W., Goethals, B., Morik, K. (eds.) ECML/PKDD 2008, Part II. LNCS (LNAI), vol. 5212, pp. 628–643. Springer, Heidelberg (2008)

ALICE@home: Distributed Framework for Detecting Malicious Sites

Ikpeme Erete[1], Vinod Yegneswaran[2], and Phillip Porras[3]

[1] Georgia Institute of Technology
ikpeme@cc.gatech.edu
[2] SRI International
vinod@csl.sri.com
[3] SRI International
porras@csl.sri.com

Abstract. Malware silently infects millions of systems every year through drive-by downloads, i.e., client-side exploits against web browsers or browser helper objects that are triggered when unsuspecting users visit a page containing malicious content. Identifying and blacklisting websites that distribute malicious content or redirect to a distributing page is an important part of our defense strategy against such attacks. However, building such lists is fraught with challenges of scale, timeliness and deception due to evasive strategies employed by adversaries. In this work, we describe alice@home, a distributed approach to overcoming these challenges and actively identifying malware distribution sites.

The growing prevalence of browser exploits and client-side attacks demands better surveillance strategies to actively blacklist malicious sites and to detect new and zero-day exploits. An approach that has been suggested is actively patrolling the Internet for sites that surreptitiously install malicious software [6]. However, scanning the Internet for miscreant sites is fraught with challenges of both scale and evasion. First, modern search engines track over a trillion web links [4] and using a virtual machine to assess the forensic impact of visiting each link requires minutes of processing time. Second, adversaries use javascript obfuscators, IP address tracking and website cloaking to evade patrolling systems. Third, most search engines do not index javascript content and simply searching on exploit features is insufficient to discover malicious sites. To address these challenges, we need to devise an *intelligent* and *Internet-scale* approach to the malicious site discovery problem.

In this paper, we present the case for a distributed architecture alice@home for tracking malicious websites. Much like, other @home projects [5], this distributed approach utilizes the idle processing cycles of desktops to crawl the web for infected sites. This enables alice@home to scale to tens of millions of pages per day assuming 1000-node network and avoid IP-tracking strategies employed by distribution sites. The incentive for participation in this network would be tangible, i.e., the ability to receive a real-time feed of malicious sites to blacklist from other paticipants.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 362–364, 2009.
© Springer-Verlag Berlin Heidelberg 2009

1 ALICE: Virtual Honey-Client

A key component of alice@home is ALICE (A Low-Interaction Crawler and Emulator), a drive-by download discovery and analysis tool that is lightweight, scalable, self-learning, easily deployable, precise, and resilient to evasion. Locating malicious scripts embedded in a page is a challenging task due to complexities such as redirects, obfuscation, multiple layers of frames and iframes, and self-modifying script code. Static analysis of these scripts are difficult since attackers typically obfuscate scripts or use string operations to conceal script contents. Therefore, we use dynamic analysis to generate and output execution traces of the script. ALICE, strips away the need to have a forensic tool or a VM running a vulnerable OS and browser by using a lightweight browser emulator able to execute script in a safe way. Scripts typically manipulate the Document Object Model (DOM) of a web page in a browser. Therefore, we emulate the DOM hierarchy by implementing a light weight browser. This browser provides the necessary support script functions, DOM hierarchal structure, safe execution environment and exposes the execution path of the script. Then we use spidermonkey (Mozilla's C implementation of a javascript engine) to execute all javascripts. Our analysis engine post-processes the output of these scripts and compares them with a dictionary of known exploits to decide malicious scripts. Its lightweight design allows a single instance of ALICE to process over 12 URLs per minute.

2 Preliminary Results

We are able to determine the malware distribution sites in the case of MDAC vulnerabilities or the type of BHO vulnerabilities that is exploited by an attacker. In the case of MDAC vulnerabilities, the location of malware distribution site is often different from the landing site, i.e., sites typically visited by users. To evade detection, these distribution sites often incorporate techniques such as non-determinism, IP tracking and fast flux to rapidly change binding IP addresses of domain names.

Processing rate. We evaluated the processing rate of ALICE relative to [2,6,1]. In the worst case ALICE is atleast 300% faster than Wang et al and in the best case it is 17% faster than Moshchuk et al 's approach with optimization. Unfortunately, the technique by [2] affects the detection capability of their system given that some of the steps used by distribution sites to evade detection. One of the distinguishing features of ALICE versus PhoneyC [3], a similar virtual honeyclient, is that PhoneyC takes an average of 2.1 hours to process URL.

Detection. In our initial testing of 35,000 urls, we detected 1294 drive-by download sites. Our initial focus was on attacks that exploit MDAC vulnerabilities. These sites linked to 33 unique distribution sites hosting malicious binaries. All 33 distribution sites infrequently infected a host by using IP tracking. Nonetheless, we were able to detect these sites.

References

1. Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D., Levy, H.M.: Spyproxy: Execution-based detection of malicious web content. In: 16th USENIX Security Symposium (August 2007)
2. Moshchuk, A., Bragin, T., Gribble, S.D., Levy, H.M.: A crawler-based study of spyware on the web. In: Network and Distributed System Security Symposium (February 2006)
3. Nazario, J.: Phoneyc: A virtual client honeypot. In: 2nd USENIX Workshop on Large-Scale and Emergent Threats, Boston, MA (April 2009)
4. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iframes point to us. In: 17th USENIX Security Symposium (2008)
5. Anagnostakis, K.G., Antonatos, S., Markatos, E.P.: Honey@home: A new approach to large-scale threat monitor. In: 5th ACM Workshop on Recurring Malcode (2007)
6. Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In: Network and Distributed System Security Symposium (NDSS), San Diego, CA (2006)

Packet Space Analysis of Intrusion Detection Signatures

Frédéric Massicotte

Communications Research Centre Canada
Ottawa, Ontario, Canada

Abstract. It is often the case that more than one signature is triggered on a given group of packets, depending on the signature database used by the IDS. For performance reasons, network IDSs often impose an *alert limit* (i.e., they restrict) on the number of signatures that can be triggered on a given group of packets. Thus, it is possible that some signatures that should be triggered to properly identify attacks are not verified by the IDS and lead to an IDS Evasion attack. In this poster, we introduce the concept of packet space analysis as a solution to address these problems.

IDS signatures sometimes overlap (i.e., they partially specify the same group of packets) and sometimes they even completely include some other signatures (i.e., a packet that triggers one signature will always trigger some other). As an illustration, here are Snort signature 1672 and signature 336.

– alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP CWD ~ attempt"; flow:to_server,established; content:"CWD"; nocase; pcre:"/^CWD\s+~/smi"; sid:1672; rev:11;)
– alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP CWD ~ root attempt"; flow:to_server,established; content:"CWD"; nocase; content: "~root"; distance:1; nocase; pcre:"/^CWD\s+~root/smi"; sid:336; rev:10;)

Here, we can see that signature 1672 includes signature 336 because the constraints (specified by the plug-ins) of signature 336 are strictly more restrictive than the constraints of signature 1672.

The inclusions and intersections between signatures in an IDS database could have security implications. (1) It is important to know if a group of signatures prevents other signatures from being triggered because of an alert limit.[1] This type of attack could lead to an IDS evasion attack. (2) The alert limit cannot be arbitrary and should be identified using a systematic analysis. Increasing this alert limit is a solution to prevent an IDS evasion attack, but it could decrease performance and lead the IDS to miss packets (or attacks). Identifying the optimal alert limit (i.e., the maximum number of signatures intersecting) is

[1] Snort 2.8.4 (released in 2009), has an alerting limit of 3 signatures.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 365–366, 2009.
© Springer-Verlag Berlin Heidelberg 2009

crucial to identifying the weaknesses of a signature database. Our research on packet space analysis of an IDS signature database (i.e., analyzing signatures for inclusions and intersections) could be used to address (1) and (2).

To analyze the packet space of an IDS signature database, we propose an approach that uses set theory. We convert signatures into sets of packets. Thus, standard set theory operations such as $\cup$, $\cap$ and $\setminus$ can be used. Let $P_{S_i^d}$ be the set of packets that triggers signature S_i of IDS d. Thus, S_i^d *includes* S_j^d iff $P_{S_i^d} \subseteq P_{S_j^d}$. Similarly, S_i^d *intersects* S_j^d iff $P_{S_i^d} \cap P_{S_j^d}$.

This approach cannot be used without a representation of $P_{S_i^d}$ allowing the computation of $\cup$, $\cap$ and $\setminus$. In the case of Snort, to calculate $P_{S_i^d}$ from S_i^d, the plug-ins related to protocol header fields are converted into ranges of values with a min and a max and associated with a protocol header field. It would be inappropriate to convert the Snort payload plug-ins into ranges of values. For example, converting the following `pcre` regular expression `/^CWD\s+~root/smi` into a range of values is not the proper model to use because its representation into ranges of values is very complex and payload plug-ins can overlap (i.e., specify the same bytes in the packet payload) such as signature 1672 and 336. To address this situation, we convert each Snort payload plug-in into one finite state automaton (FSA) that represents the constraints on the packet payload. As a result, $P_{S_i^d}$ is represented as a FSA for the packet payload and a set of ranges R_f^{Π} where f is a field of protocol Π. The ranges and the FSA can then be used with $\cup$, $\cap$ and $\setminus$ operators.

Based on this, we developed an IDS Signature Space Analyzer (IDS-SSA) to identify problems in a signature database. We obtained interesting results using a prototype version of IDS-SSA for signature inclusions on 12 Snort signature databases downloadable without registration.[2] The intersection calculation is currently not implemented. In Snort 2.4.0[3], we identified three pairs of equal signatures (i.e., $S_i^d = S_j^d$), 266 inclusion sequences of length two (i.e., $S_i^d \subset S_j^d$) and two inclusion sequences of length three. For this case study, we did not identify an inclusion sequence longer than four. These results suggest redundancies between signatures. They also suggest situations that could be exploited by an attacker (e.g., IDS evasion) as well as potential problems or errors (e.g., signatures that are equal) in the signature database.

The next step is to implement the signature intersection analysis to identify whether or not there are sets of intersecting signatures with size greater than the alert limit and to calculate (if possible) or approximate the optimal alert limit. Moreover, the IDS-SSA could also be used to compare signatures between different signature databases.

[2] 1.8.6, 1.8.7, 1.9.0, 1.9.1, 2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.3.1, 2.3.2, 2.3.3 and 2.4.0.

[3] Last version of the signature database downloadable without registration.

Traffic Behaviour Characterization Using NetMate

Annie De Montigny-Leboeuf, Mathieu Couture, and Frederic Massicotte

Communications Research Centre Canada (CRC) , Ottawa, ON, Canada
{annie.demontigny-leboeuf, mathieu.couture,
frederic.massicotte}@crc.gc.ca

Abstract. Previous studies have shown the feasibility of deriving simple indicators of file transfers, human-interactivity, and other important behavioural characteristics. We are proposing a practical implementation and use of such indicators with NetMate. In its current state as a work in progress, our extended version of NetMate will already be of interest to network security practitioners conducting incident analysis. The tool can be used to post-process traffic traces containing suspicious flows in order to obtain a behavioural description of the incident and surrounding traffic activities. With further development, the approach has great potential for other use cases such as intrusion detection, insider threat detection, and traffic classification.

The majority of current network monitoring tools rely on well-known port numbers and/or payload analysis to identify the network applications. While payload analysis is more reliable than port numbers, decoding is not always possible due to encryption or obfuscation, for instance.

Previous research studies have shown that classifying network traffic according to flow behaviour is feasible and promising [1][2][3][4]. Despite relevant studies, we are not aware of traffic characterization solutions of this type being used in practice by network security professionals.

Our proposed implementation is designed with practitioners in mind. Installation and use are simple and similar to other common packet processing tools such as tcpdump and snort. The output is intuitive and can shed light on traffic activities under investigation.

This implementation is based on [4], in which the flow features (metrics) have discriminative power and also provide insight into the traffic behaviour. The set of flow features, inspired by the the work of Paxson [5], includes indicators of interactivity (human control), conversation, transaction, data transfer, and many other important behavioural characteristics. The analysis is confined to headers at the network and transport layers and thereby does not depend on access to payload.

We are extending NetMate [6], an existing open-source packet processing framework. NetMate includes two types of modules: *Packet Processing Modules* designed to implement different metrics, and *Export Modules* that implement different output formats. Our flow features are being implemented in *Packet Processing Modules*, and our rule engine that describes and recognizes the flows is being implemented as an *Export Module*.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 367–368, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Our *Export Module* compares the measured flow metrics against a configurable set of rules, which are stored in files. When NetMate is configured to use our new modules and set of rules, the output file it produces consists of one line per flow (i.e. per TCP/UDP 5-tuple session). Here is a sample output:

```
yyyy-mm-dd hh:mm:ss, 4, 6, xx.xx.xx.xx, yy.yy.yy.yy,
35573, 22, directional-bkwd, persistent, human-
keystroke-fwd, no-bulk-transfer, free-transmitrate,
blockcipher-encrypted, , , , , , ssh, , , , , , , , ,
```

Two different types of rules are used to produce the output: *description rules* and *recognition rules*. In the example above, the *description rules* produce the portion of the output in underlined characters, while the text in *italics* is due to the *recognition rules*. In this example, one and only one of the protocol-profiles, *ssh*, matched. The first portion of the output is simply composed of the date, time, and the flow key (IP version, proto ID, IP addresses, and port numbers).

Current State and Future Work. We need to evaluate performance on streaming traffic. We also need to evaluate the detection rates and false positive rates of the rules implemented to date. These rules come from [4] and have been implemented following their original specifications, which were derived manually. The use of machine-learning techniques [1][2][3] to derive *recognition* and *description rules* will be considered in future work. The *description rules* currently implemented provide sensible and insightful overview of the traffic activities. We believe that the description capability provides in itself much added value. While the implemented *recognition rules* attempt to identify protocols, these rules can also be defined to recognize general types of traffic of interest to an analyst.

Despite its early development stage, we believe that our current version, as is, can be useful to security analysts in the context of supporting network incident analysis. We are therefore working on the first release of our extended NetMate, which we expect will be available by September 2009.

References

1. Kim, H., Claffy, K., Fomenkov, M., Barman, D., Faloutos, M., Lee, K.: Internet Traffic Classification Demystified: Myths, Caveats, and Best Practices. In: ACM CoNEXT (2008)
2. Alshammari, R., Zincir-Heywood, A.N.: A preliminary Performance Comparison of Two Feature Sets for Encrypted Traffic Classification. In: IEEE CISIS (2008)
3. Williams, N., Zander, S., Anmitage, G.: A preliminary Performance comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. ACM SIGCOMM CCR 36(5) (2006)
4. De Montigny-Leboeuf, A.: Flow Attributes for Use in Traffic Characterization., CRC Technical Note CRC-TN-2005-003 (2005)
5. Zhang, Y., Paxson, V.: Detecting Backdoors. In: USENIX Security Symposium (2000)
6. NetMate, `http://www.ip-measurement.org/tools/netmate/` (last accessed June 2009)

On the Inefficient Use of Entropy for Anomaly Detection*

Mobin Javed[1], Ayesha Binte Ashfaq[1], M. Zubair Shafiq[2], and Syed Ali Khayam[1]

[1] National University of Sciences & Technology, Islamabad, Pakistan
[2] nexGIN RC, FAST National University, Islamabad, Pakistan
{mobin.javed,ayesha.ashfaq,ali.khayam}@seecs.edu.pk,
zubair.shafiq@nexginrc.org

Abstract. Entropy-based measures have been widely deployed in anomaly detection systems (ADSes) to quantify behavioral patterns. The entropy measure has shown significant promise in detecting diverse set of anomalies present in networks and end-hosts. We argue that the full potential of entropy-based anomaly detection is currently not being exploited because of its inefficient use. In support of this argument, we highlight three important shortcomings of existing entropy-based ADSes. We then propose efficient entropy usage – supported by preliminary evaluations – to mitigate these shortcomings.

1 Entropy Limitations and Countermeasures

1.1 Feature Correlation Should Be Retained

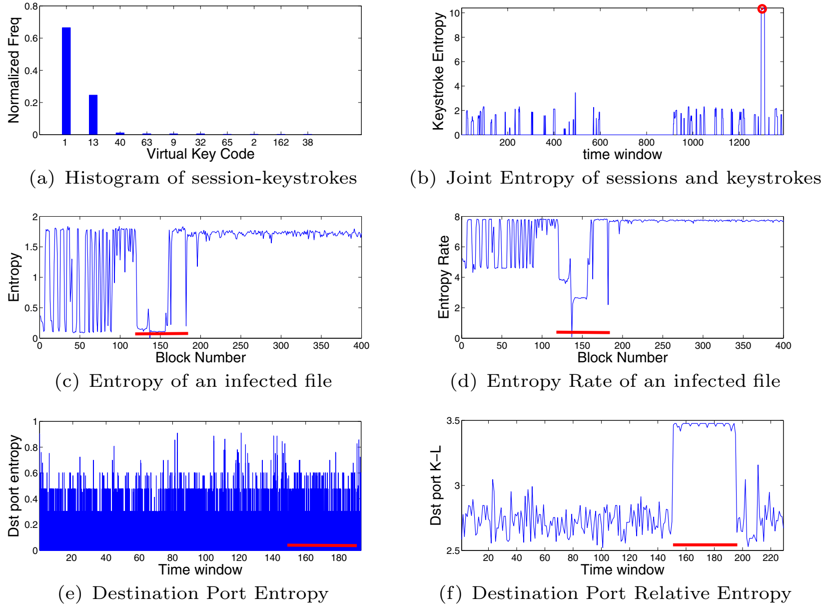
Current ADSes perform entropy analysis on *marginal distributions* of features. In general, significant correlation exists across traffic and/or host features which is not being leveraged by these ADSes. As a proof-of-concept example, we propose to detect malicious network sessions by noting that the histogram of keystrokes which are used to initiate network sessions is skewed [see Fig. 1(a)] and perturbation in this metric can easily reveal the presence of an anomaly; network traffic and keystroke data were collected before and after infecting a human-operated computer with the low-rate `Rbot-AQJ` worm. While analyzing the entropies of the marginal keystroke distribution and/or the marginal session distribution is clearly not useful, Fig. 1(b) shows that quantifying these features using joint (session-keystroke) entropy can easily detect anomalous activity.

1.2 Spatial/Temporal Correlation Should Be Retained

Another limitation of the entropy measure is its inability to take spatial/temporal correlation of benign patterns into account. Such correlations can prove useful in the detection of subtle anomalies. For instance, Fig. 1(c) shows the block-wise

* This work is supported by the Pakistan National ICT R&D Fund.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 369–370, 2009.
© Springer-Verlag Berlin Heidelberg 2009



(a) Histogram of session-keystrokes

(b) Joint Entropy of sessions and keystrokes

(c) Entropy of an infected file

(d) Entropy Rate of an infected file

(e) Destination Port Entropy

(f) Destination Port Relative Entropy

Fig. 1. Examples to support the limitations of the current use of entropy

(block size = 1KB) entropy of a `PDF` file which is infected by an embedded executable malware. It is evident that entropy is unable to provide clear perturbations required for detection. On the other hand, entropy rate [Fig. 1(d)], which models and accounts for the spatial/temporal correlation, provides very clear perturbations at the infected file blocks; entropy rate quantifies the average entropy of conditional distributions.

1.3 Randomness Quantification Is Not Enough

Entropy cannot distinguish between differing distributions with the same amount of uncertainty; e.g., entropy of the normalized distribution of a source producing 90 packets on port 80 and 10 packets on port 21 is the same as a source producing 900 packets on port 6666 and 100 packets on port 6667. Thus anomalies which do not perturb randomness go undetected. Fig. 1(e) shows a case where the Blaster worm cannot be detected in the destination port entropy time-series. This limitation arises due to the fact that entropy does not take the individual port numbers into account. It is, therefore, important to perform a symbol-by-symbol comparison between benign and observed distributions. This can be achieved by computing the *relative entropy* of the distributions. Fig. 1(f) shows that K-L divergence time series of destination port is perturbed due to the presence of `Blaster` worm.

Browser-Based Intrusion Prevention System

Ikpeme Erete

Georgia Institute of Technology
ikpeme@cc.gatech.edu

Abstract. This work proposes a novel intrusion prevention technique that leverages information located in the browser in order to mitigate client-side web attacks such as login cross-site request forgery, session hijacking, etc. The browser intrusion prevention system enforces a new fine-grained policy, which complements the same-origin policy, that restricts interaction between authenticated and unauthenticated regions of a page or its associated stored data objects. The browser intrusion prevention system monitors page interactions that occur through script processing or URL fetches. The outcome of this technique is a system that can prevent attacks that are perpetuated by exploiting a user's browser into making malicious request.

1 Motivation

The Hypertext Transfer Protocol (HTTP) is a generic, stateless protocol, [1] that maintains no information on a connection between a host and a server. In order to identify a returning host, web applications use interfaces provided by servers to implement a session management system. By using HTTP response header to set state information on a host's browser and subsequently associating the state information with a particular host or user at the server side, a web application can keep track of user's activities. This enables a web application on the server to identify a returning user by extracting the necessary state parameters from the HTTP request that was generated and sent by the user's browser on subsequent visits. It also uses this technique to manage authenticated sessions; thereby eliminating the need to constantly send authentication credentials such as username and password, on every request.

Since browsers automatically present "what it knows" to the server as a form of authentication (i.e. validation or identification) attackers have developed ways of leveraging this to exploit users by exploiting the browser into making malicious requests on users' behalf. Attacks such as cross site request forgery, surf jacking, dynamic pharming [4], have been shown to use this technique to mount a successful exploit. Most solutions or proposed solutions against such attacks are focused at the server side [3], while others recommend better browsing habits or changes to HTTP protocol [2]. Though these solutions are viable, they are ineffecient since they require the use of additional resources or are long term solutions that may be implemented in newer browsers. While in the short term, users are continuously exposed to these attacks. To mitigate these attacks, this work takes advantage of information stored in browsers to provided the much needed security.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 371–373, 2009.
© Springer-Verlag Berlin Heidelberg 2009

2 Browser-Based Intrusion Prevention System

Websites provide user authentication to protect user's sensitive data from unauthorized accesses. Therefore, successfully exploiting such sites provide high remuneration to an attacker since these sites might contain valuable data. Furthermore, this task is made easier since browsers maintain all the necessary credentials pertaining to any session including an authenticated session.

The proposed client-side browser intrusion prevention system consist of a policy and an enforcement mechanism. The policy states that an unauthenticated page region cannot read or write to an authenticated page region or to its stored private data. This is a finer grain policy to the same-origin policy, SOP, but it is not a substitute to SOP since different regions could belong to the same-origin. For example, many web sites use "https" connection during authentication but immediately revert to "http" connection once the authentication is successful. Consequently, the policy description and enforcement ensures that the page region changes once authentication is successful but the origin remains the same. Therefore, the described policy complements SOP.

The enforcement mechanism detects a change in region when it observes that a username and password is entered by a user. After which, it associates this information with the corresponding http-response from the server. If a success response, 200 OK, is received, the enforcement mechanism labels the page as authenticated. When a page and its components (e.g., frames, iframes) are placed in regions, all cross-regional accesses are monitored by the enforcement mechanism.

By monitoring these accesses to authenticated regions it is possible to identify and prevent an attack while accesses to unauthenticated region are permitted with no checks. In order to properly analyze these accesses all http-request is monitored and mediated. Each http-request header is examined and all credentials such as GET/POST destination, session ids, referrer, and cookies are extracted. Using the retrieved information, the enforcement mechanism determines the source of the request. Subsequently, it deduces whether the source of the request is authenticated or unauthenticated. The page region is verified in accordance with the proposed policy. If the policy is violated, the request is blocked, otherwise the enforcement mechanism inspects the session credentials. The session credentials are inspected for any policy violation and if one is found, the credentials are filtered. Filtering the session credentials involve eliminating those credentials that violate the policy from the request, permitting the request to go through. In likewise manner, page or stored data accesses via scripts are also mediated in accordance with the policy.

References

1. Hypertext transfer protocol –http/1.1
2. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: ACM Conference on Computer and Communications Security (2008)

3. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: Proceedings of the Second IEEE Conference on Security and Privacy in Communications Networks (SecureComm), pp. 1–10 (2006)
4. Karlof, C., Shankar, U., Tygar, J.D., Wagner, D.: Dynamic pharming attacks and locked same-origin policies for web browsers. In: ACM Conference on Computer and Communications Security, pp. 58–71 (2007)

Using Formal Grammar and Genetic Operators to Evolve Malware

Sadia Noreen[1], Shafaq Murtaza[1], M. Zubair Shafiq[2], and Muddassar Farooq[2]

[1] FAST National University, Islamabad, Pakistan
[2] nexGIN RC, FAST National University, Islamabad, Pakistan
{sadia.noreen,shafaq.murtaza}@nu.edu.pk,
{zubair.shafiq,muddassar.farooq}@nexginrc.org

Abstract. In this paper, we leverage the concepts of *formal grammar* and *genetic operators* to evolve malware. As a case study, we take *COM infectors* and design their formal grammar with production rules in the BNF form. The chromosome (abstract representation) of an infector consists of genes (production rules). The code generator uses these production rules to derive the source code. The standard genetic operators – crossover and mutation – are applied to evolve population. The results of our experiments show that the evolved population contains a significant proportion of valid COM infectors. Moreover, approximately 7% of the evolved malware evade detection by COTS anti-virus software.

1 Evolutionary Malware Engine: An Empirical Study

Malware writers have developed *malware engines* which create different variants of a given malware – mostly by applying packing techniques. The developed variants essentially have the same functionality and semantics. In contrast, our methodology targets to create "new" malware. It consists of three phases: (1) design a formal grammar for malware and use it to create an abstract representation, (2) use standard genetic operators – crossover and mutation, and (3) generate assembly code from the evolved abstract representation.

The working principle of the proposed COM infector evolution framework is shown in Fig. 1. In the first step, it analyzes the source code of an infector and maps it to the production rules – defined in the formal grammar – to generate its chromosome. This step is initially done for 10 infectors (source code is obtained from [1]); resulting in a population of 10 chromosomes. We then apply genetic operators – crossover and mutation – to the population. Intuitively speaking, all individuals will not be legitimate infectors after genetic operators are applied. To test this hypothesis, we have a code generation unit which accepts these chromosomes and produces assembly code for them. Finally, we present the evolved malware to well-known COTS anti-virus products to check if the evolved infectors can evade detection.

We have observed that the evolved infectors fall into one of the three categories: (1) COM infectors which have turned benign, (2) COM infectors which

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 374–375, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Fig. 1. Architecture of COM infector evolution framework

```
.model small
.code

FNAME EQU 9EH
ORG 100H

START:
mov ah, 4EH
mov dx, OFFSET COM_FILE          -> Search First
int 21H

SEARCH_LP:
jc DONE
mov ax, 3D01H
mov dx, FNAME                    -> File Open
int 21H

xchg ax, bx
mov ah, 40H
mov cl, 42                       -> File Write
mov dx, 100H
int 21h

mov ah, 3EH                      -> File Close
int 21H

mov ah, 4FH                      -> Search Next
int 21h
jmp SEARCH_LP
DONE:
ret

COM_FILE DB "*.COM",0
END START
```

Fig. 2. Code of mini44

```
1) <Virus>::=<SF> <FO> <FW> O(<FC>) O(<SN>)
2) <SF>::=<Routine>
3) <FO>::=<Routine>
4) <FW>::=<Routine>
5) <FC>::=<Routine>
6) <SN>::=<Routine>
7) <Routine>::=N(<Statement>) | ε
8) <Statement>::=<DataMov> | <PrcsCntl>
9) <DataMov>::=<Move> | <Xchg>
10) <Move>::="mov" <lst> "," <2nd>
11) <lst>::=<Reg8> | <Reg16> | <Segreg>
12) <2nd>::=<Reg 8> | <Reg16> | <Segreg> | <Mem> | <Imm>
13) <Xchg>::="xchg" <Reg> "," <Reg>
14) <Mem>::=<Identifier> | "OFFSET" <Identifier>
15) <Imm>::=<Digit> #(<Digit>) | <Hex_Digit> #(<Hex_Digit>) "H"
16) <Reg 8>::= "AH"| "AL"| "BL"| "BH"| "CL"| "CH"| "DH"| "DL"
17) <Reg16>::= "AX"| "BX"| "CX"| "DX"
18) <Segreg>::= "CS" | "ES"| "SS"| "FS"| "DS"| "GS"| "SI"
19) <Identifier>::=<Nondigit> #(<Nondigit> | <Digit>)
20) <Nondigit>::=[A-z a-z]
21) <Digit>::=[0-9]
22) <Hex_digit>::=[0-9 a-f A-F]
23) <PrcsCntl>::= "int 21H"
```

Fig. 3. BNF of COM infectors

are detected by anti-virus but as a different type than that of initial 10 infectors, and (3) unknown variants of COM infectors which have successfully evaded the detection mechanism. We manually execute the last category of the infectors on Windows XP machine to check if the evolved infectors truly do the damage. Our initial findings show that about 52% of evolved infectors have become benign; 41% are detected but with new names that are not included in the initial population; while remaining 7% still do their destructive job but remain undetected. The last category of infectors have achieved stealthiness in the true sense.

We now take an example of a simple `mini44` malware (see Fig. 2) to explain the evolution procedure. The common routines – Search First, Copy, Search Next – are labeled in Fig. 2. *Search First* routine searches for the first COM file in the current directory and it then opens it. After opening the file, the malware writes its code into the victim file and the file is closed. The next victim COM file is searched in *Search Next* function. Once our engine will read instruction *mov ah, 4EH* of `mini44`, it will lookup for the production rules that match with this instruction. The production rules are given in Fig. 3. The genotype of the instruction *mov ah, 4EH* may consist of following production rules: 1-2-7-8-9-10-11-12-16-15-22. In a similar fashion, the genotype of each insturction/routine in COM infector is generated. When we want to produce a new individual, we take abstract representation of two infectors and use crossover and mutation operators to evolve new individuals. Finally, the code generator does the reverse mapping to generate the source code of the evolved infector.

References

1. Virus Source Code Database, VSCDB, http://www.totallygeek.com/vscdb/

Method for Detecting Unknown Malicious Executables

Boris Rozenberg[1], Ehud Gudes[1], Yuval Elovici[2], and Yuval Fledel[2]

[1] Deutche Telekom Laboratories at BGU and Department of Computer Science,
[2] Deutche Telekom Laboratories at BGU and Department of Information System Engineering,
Ben Gurion University, Beer Sheva 84105, Israel

Abstract. We present a method for detecting new malicious executables, which comprises the steps of: (a) in a training phase, finding a collection of system call sequences that are characteristic only to malicious files, and storing said sequences in a database; (b) in a runtime phase, for each running executable, continuously monitoring its issued run-time system calls and comparing with the stored sequences within the database, and when a match is found, declaring said executable as malicious.

1 Introduction and Related Works

Detection of known malicious executables is typically performed using signature-based techniques. The main disadvantage of these techniques is the inability to detect totally new malicious executables. The main prior art approach for detecting new malicious executables is to employ machine learning and data mining for the purpose of creating a classifier that is able to distinguish between malicious and benign executables statically [1]. The main drawback of the above approach is its inability to deal with obfuscated or encrypted files, that results in false alarms. In this paper we introduce a novel technique for the real-time detection of new malicious executables that follows dynamic analysis approach.

2 Our Method

Fig. 1 is a flow diagram illustrating the process for detecting malicious executables. During the training phase 101, which is performed off-line, an "M determining module" 102 operates to determine sequences of system calls that are characteristic only to malicious executables. This module produces an "M database" 103 which forms an input data to comparator 104. During the runtime monitoring phase 105, comparator 104 continuously receives inputs relating to the system calls that are issued by the currently running executables, compares separately for each running program the sequence of system calls that it issues, with each of the sequences stored in the M database. If a match is found with one or more of the M-sequences, a program is declared as malicious and can be terminated. Otherwise, as long as no such an alert signal is issued, a running file is considered as begin.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 376–377, 2009.
© Springer-Verlag Berlin Heidelberg 2009

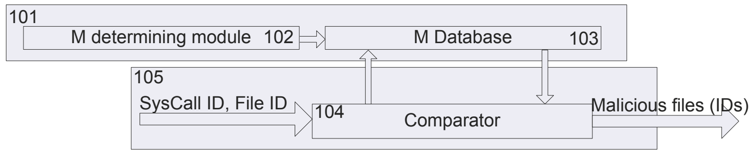


Fig. 1. Process for detecting malicious executables - flow diagram

Fig. 2 describes a training phase process for determining the database of M-sequences. The process comprises accumulation of *n* malicious and *m* benign executables. In steps 201 and 202, each executable is executed, and its runtime sequence of system calls is recorded. The result is *Mr* dataset which contains *n* records and *Br* dataset which contains *m* records. In step 203, a set *S* of all frequent sequences in *Mr* is determined by applying the SPADE algorithm [2]. It should be noted that each *si* in the found set *S* may contain wildcards. In step 204, for each sequence *si* in *S* the process checks whether the sequence *si* appears within any of the sequences included within the dataset *Br*. If it is, that means that *si* is not a suitable sequence for the purpose of determining malicious executables according to our method. The output from step 204 is therefore a minimal set *Sm*, which includes only those sequences from *S* that do not appear in any of the sequences of *Br*, and therefore are characteristic to only malicious executables.

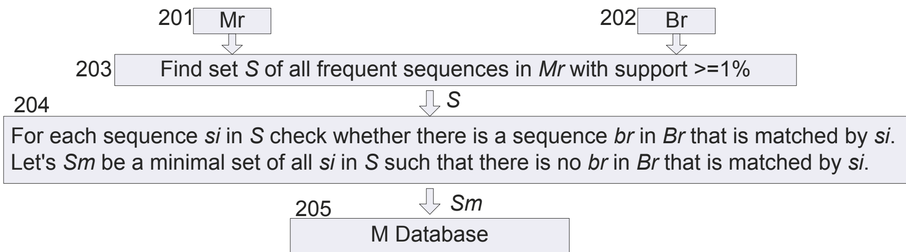


Fig. 2. Training phase process

3 Evaluation

We ran 3-fold cross-validation on 700 malicious and 700 benign files. We have discovered 28 characteristic sequences that match 87% of malicious executables, with 7% false alarms.

References

1. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: 10th ACM SIGKDD international conference on knowledge discovery and data mining, pp. 470–478. ACM Press, New York (2004)
2. Zaki, M.G.: Efficient Algorithm for Mining Frequent Sequences. Machine Learning 42, 31–60 (2001)

Brave New World: Pervasive Insecurity of Embedded Network Devices

Ang Cui, Yingbo Song, Pratap V. Prabhu, and Salvatore J. Stolfo

Intrusion Detection Systems Lab,
Columbia University, NY, USA
{ang,yingbo,pvp2105,sal}@cs.columbia.edu

Abstract. Embedded network devices have become an ubiquitous fixture in the modern home, office as well as in the global communication infrastructure. Devices like routers, NAS appliances, home entertainment appliances, wifi access points, web cams, VoIP appliances, print servers and video conferencing units reside on the same networks as our personal computers and enterprise servers and together form our world-wide communication infrastructure. Widely deployed and often misconfigured, they constitute highly attractive targets for exploitation. In this study we present the results of a vulnerability assessment of embedded network devices within the world's largest ISPs and civilian networks, spanning North America, Europe and Asia. The observed data confirms the intuition that these devices are indeed vulnerable to trivial attacks and that such devices can be found throughout the world in large numbers.

Keywords: Router insecurity, network webcams, print servers, embedded device management interface exploitation, default password.

1 Introduction

Embedded network devices perform specific functions like routing, file storage etc. Competition among manufacturers demand that these products be produced with minimal time to market at the lowest cost possible. These common commodity products are often implemented without security in mind and reside on the same networks as general purpose computers, making them attractive targets for exploitation. Once compromised, communication devices like routers, voip appliances, and video conferencing units can be used to quietly intercept and alter the traffic they carry. Nearly all embedded network devices contain a network interface which can be used to perform layer-2 and layer-3 attacks on the rest of the network. Since host based protection schemes for embedded devices generally do not exist today and network based protection schemes (802.1X etc) often intentionally exclude such devices administratively, exploitation and root-kitting[1] of these devices proves to be very advantageous to the attacker.

[1] The companion paper to this field survey will detail Doppelgänger; a semi-virtualized exploitation method for root-kitting heterogeneous embedded devices in a device and operating system agnostic manner.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 378–380, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Table 1. Key scan statistics thus far

Total IPs Scanned	Webservers	Telnet Servers	Devices Targeted	Vul. Devices Found
85.7 Million	1.1 Million	800 Thousand	105,357	3,847

Table 2. Vulnerability rate by device class

Enterprise Devices	VOIP Devices	Consumer Devices
2.46%	19.21%	41.62%

Fig. 1. Linksys vul. distribution

JPN	CAN	IND	KOR
75.0%	60.0%	57.1%	57.1%
HUN	AUT	NLD	USA
54.5%	50.0%	48.6%	38.5%
CZE	FRA	URY	CHN
38.5%	34.2%	18.9%	10.0%

Fig. 2. Linksys vul. by country

2 Methodology

This paper presents preliminary results from our larger communications insecurity study by scanning, on a global scale, for perhaps the simplest attack possible; publicly accessible administrative interface with default password. We targeted the largest ISP networks in North America, Europe, and Asia, scanning and cataloging popular network appliances accessible over the internet. Out of all discovered devices, we then tabulated the number of such devices which are configured with their factory default administrative passwords. This data is then broken down by device types (Linksys, Polycom, Cisco etc), device class (Consumer, Enterprise, VOIP etc) and by geographical region (Zipcodes within the US and by Country world-wide).

3 Findings

It is possible to draw several high level conclusions from the observed data. **Insecurity is pervasive world-wide**: Vulnerable devices can be found in significant numbers in all parts of the world covered by our scan. The double digit vulnerability rates suggest that a large botnet can be created by constituting only embedded network devices. [1]. **Geographical variations exist**: We found significant geographical concentrations of vulnerable devices of several types. This is undoubtedly related to the targeted markets of these devices. **Consumer devices are most vulnerable**: Looking at the vulnerability rates between consumer and enterprise devices world-wide, we see a significant difference between 45.62% versus 2.46%.

Detailed findings of our vulnerability assessment will be published in it's entirety upon the completion of the global scan.

Reference

1. DroneBL: Dd-wrt botnet, http://dronebl.org/blog/8

DAEDALUS: Novel Application of Large-Scale Darknet Monitoring for Practical Protection of Live Networks (Extended Abstract)

Daisuke Inoue[1], Mio Suzuki[1], Masashi Eto[1], Katsunari Yoshioka[2], and Koji Nakao[1]

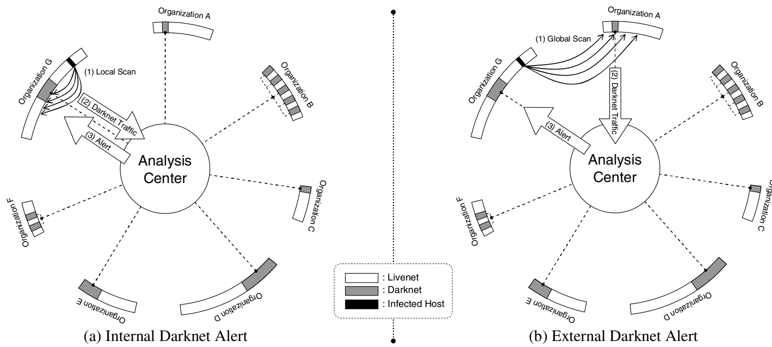
[1] National Institute of Information and Communications Technology (NICT)
[2] Yokohama National University
{dai,mio,eto}@nict.go.jp, yoshioka@ynu.ac.jp, ko-nakao@nict.go.jp

Abstract. Large-scale darknet monitoring is an effective approach to grasp a global trend of malicious activities on the Internet, such as the world-wide spread of malwares. There, however, have been a gap between the darknet monitoring and actual security operations on live networks, namely the global trend has less direct contribution to protect the live networks. Therefore, we propose a novel application of large-scale darknet monitoring that significantly contributes to the security of live networks. In contrast to the conventional method, wherein the packets received from the outside are observed, we employ a large-scale distributed darknet that consists of several organizations that mutually observe the malicious packets transmitted from the inside of the organizations. Based on this approach, we have developed an alert system called DAEDALUS (direct alert environment for darknet and livenet unified security). We present the primary experimental results obtained from the actual deployment of DAEDALUS.

Keywords: darknet monitoring, live network protection, alert system.

The Internet encounters considerable amounts of unwanted traffic and attacks, which are mostly caused by malwares. An effective approach to grasp a global trend of malicious activities such as the spread of malwares is to monitor a large-scale darknet (a set of globally announced unused IP addresses) [1,2]. There, however, have been a gap between the darknet monitoring and actual security operations on the live network (hereafter referred to as livenet), which comprises legitimate hosts, servers and network devices. For instance, although darknet monitoring can be used to inform network operators about a global increase in scan on 80/tcp, it may not ensure that any concrete security operations are carried out. This means that darknet monitoring does not significantly contribute to the protection of the livenet. Therefore, we propose a novel application of large-scale darknet monitoring that significantly contributes to the security of the livenet. In contrast to the conventional method wherein the packets received from the outside are observed, we employ a large-scale distributed darknet that consists of several organizations that mutually observe the malicious packets transmitted from the inside of the organizations. Based on this approach, we have developed an alert system called DAEDALUS (direct alert environment for darknet and livenet unified security), which is illustrated in Fig. 1.

E. Kirda, S. Jha, and D. Balzarotti (Eds.): RAID 2009, LNCS 5758, pp. 381–382, 2009.
© Springer-Verlag Berlin Heidelberg 2009



(a) Internal Darknet Alert

(b) External Darknet Alert

Fig. 1. Overview of DAEDALUS

DAEDALUS consists of an analysis center and several organizations. Each organization (hereafter referred to as org) establishes a secure channel with the analysis center and continuously forwards darknet traffic toward the center. In addition, each org registers the IP address range of its livenet to the center beforehand. We divide the darknet into two types—internal and external darknet. From the viewpoint of an org, the darknet within the org is an internal darknet, and the darknets in other orgs are external darknets.

When a malware infection occurs, e.g., in org G in Fig. 1, and the infected host starts scanning the inside of the org, including the internal darknet (Fig. 1 (a)), the analysis center can detect the infection on the basis of the match between the source IP address of darknet traffic from the org G and the preregistered livenet IP address. The analysis center then sends an internal darknet alert to org G. When the infected host starts scanning the outside, including the external darknet in org A (Fig. 1 (b)), the analysis center can detect the infection in the same above-mentioned manner. The analysis center then sends an external darknet alert to org G. The alerts include information on the IP address of the infected host, protocol, source/destination ports, duration of attack, and analysis results, if any.

We are conducting a trial of DAEDALUS by using the darknet resources of nicter [3]. With a /16 mixed network of livenet and darknet (as the preregistered livenet and internal darknet) and a pure/16 darknet (as the external darknet), over 2,500 internal darknet alerts and 9 external darknet alerts were issued in a month (July 2008), some of the alerts triggered actual security operations in the org that owns the livenet.

References

1. Bailey, M., et al.: The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In: 12th Annual Network and Distributed System Security Symposium (2005)
2. Moore, D.: Network Telescopes: Tracking Denial-of-Service Attacks and Internet Worms around the Globe. In: 17th Large Installation Systems Administration Conference (2003)
3. Inoue, D., et al.: Nicter: An Incident Analysis System toward Binding Network Monitoring with Malware Analysis. In: 2008 WOMBAT Workshop on Information Security Threats Data Collection and Sharing. IEEE Computer Society Press, Los Alamitos (2008)

Author Index

Abdelnur, Humberto 182
Antonatos, Spiros 265
Ashfaq, Ayesha Binte 369

Bartoš, Karel 61
Beyah, Raheem 304
Bolzoni, Damiano 1

Caballero, Juan 161
Chandrashekar, Jaideep 326
Chen, Songqing 244
Chiueh, Tzi-cker 101
Copeland, John 304
Couture, Mathieu 367
Cretu-Ciocarlie, Gabriela F. 41
Cui, Ang 378

Debar, Hervé 81
Demay, Jonathan-Christofer 348
De Montigny-Leboeuf, Annie 367

Eidenbenz, Stephan 202
Elovici, Yuval 357, 376
Engel, Thomas 61
Erete, Ikpeme 362, 371
Estan, Cristian 284
Etalle, Sandro 1
Eto, Masashi 381

Fang, Bin-Xing 346
Farooq, Muddassar 121, 224, 374
Festor, Olivier 182
Filiol, Eric 81
Fledel, Yuval 376
François, Jérôme 182
Fusenig, Volker 61

Galli, Emanuele 202
Gao, Debin 142
George, Laurent 355
Giroire, Frederic 326
Griffin, Kent 101
Grill, Martin 61
Gudes, Ehud 376
Guo, Li 346
Guyet, Thomas 359

Hartel, Pieter H. 1
Hu, Xin 101
Hulboj, Milosz Marian 353

Inoue, Daisuke 381
Ioannidis, Sotiris 265

Jacob, Grégoire 81
Javed, Mobin 369
Jha, Somesh 284
Jurga, Ryszard Erazm 353

Kanonov, Uri 357
Khayam, Syed Ali 224, 369
Knapskog, Svein J. 359
Kruegel, Christopher 21

Lee, Wenke 350
Li, Peng 142
Li, Yang 346
Liang, Zhenkai 161
Lin-Qi 346
Liu, Lei 244
Liu, Xiang-Tao 346
Locasto, Michael E. 41
Lu, Long 350
Luchaup, Daniel 284

Maggi, Federico 21
Markatos, Evangelos P. 265
Massicotte, Frédéric 365, 367
Mé, Ludovic 355
Mirza, Fauzan 121
Murtaza, Shafaq 374

Nakao, Koji 381
Noreen, Sadia 374

Papagiannaki, Dina 326
Pěchouček, Michal 61
Polychronakis, Michalis 265
Poosankam, Pongsin 161
Porras, Phillip 350, 362
Prabhu, Pratap V. 378

Rehák, Martin 61
Reiter, Michael K. 142
Robertson, William 21
Rozenberg, Boris 376

Schneider, Scott 101
Schooler, Eve 326
Shabtai, Asaf 357
Shafiq, M. Zubair 121, 369, 374
Shahzad, Muhammad 224
Smith, Randy 284
Song, Dawn 161
Song, Yingbo 378
Staab, Eugen 61
State, Radu 182
Stavrou, Angelos 41
Stiborek, Jan 61
Stolfo, Salvatore J. 41, 378
Suzuki, Mio 381

Tabish, S. Momina 121
Taft, Nina 326
Totel, Éric 348
Tronel, Frédéric 348

Vasiliadis, Giorgos 265
Viet Triem Tong, Valérie 355
Vigna, Giovanni 21

Wang, Wei 359

Xuan, Chaoting 304

Yan, Guanhua 202, 244
Yegneswaran, Vinod 350, 362
Yoshioka, Katsunari 381

Zahid, Saira 224
Zhang, Xinwen 244