Ulrich Flegel
Evangelos Markatos
William Robertson (Eds.)

LNCS 7591

# Detection of Intrusions and Malware, and Vulnerability Assessment

9th International Conference, DIMVA 2012
Heraklion, Crete, Greece, July 2012
Revised Selected Papers

Lecture Notes in Computer Science 7591

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

Ulrich Flegel  Evangelos Markatos
William Robertson (Eds.)

# Detection of Intrusions and Malware, and Vulnerability Assessment

9th International Conference, DIMVA 2012
Heraklion, Crete, Greece, July 26-27, 2012
Revised Selected Papers

Springer

Volume Editors

Ulrich Flegel
HFT Stuttgart, Department C
Schellingstr. 24, 70174 Stuttgart, Germany
E-mail: ulrich.flegel@hft-stuttgart.de

Evangelos Markatos
Foundation for Research and Technology – Hellas (FORTH)
Department of Computer Science
100 Plastira Ave, Vassilika Vouton, 70013 Heraklion, Crete, Greece
E-mail: markatos@ics.forth.gr

William Robertson
Northeastern University
College of Computer and Information Science
360 Huntington Ave, Boston, MA 02115, USA
E-mail: wkr@ccs.neu.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-37299-5 e-ISBN 978-3-642-37300-8
DOI 10.1007/978-3-642-37300-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013934265

CR Subject Classification (1998): K.6.5, D.4.6, K.4.4, D.2, C.2, C.5.3

LNCS Sublibrary: SL 4 – Security and Cryptology



*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

On behalf of the Program Committee, it is our pleasure to present to you the proceedings of the $9^{th}$ GI International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). Each year, DIMVA brings together international experts from academia, industry, and government to present and discuss novel security research. DIMVA is organized by the Special Interest Group Security – Intrusion Detection and Response (SIDAR) of the German Informatics Society (GI).

The DIMVA 2012 Program Committee received 44 submissions from a diverse set of countries. All submissions were carefully reviewed by Program Committee members and external experts according to the criteria of scientific novelty, technical quality, and practical impact. The final selection took place at the Program Committee meeting held on April 5, 2012, at the University of Bonn. Ten full papers and four short papers were selected for presentation at the conference and publication in the proceedings. The conference took place July 26–27 at the Astoria Capsis Hotel in Heraklion, Crete. The program featured both theoretical and practical research results grouped into five sessions spanning malware, intrusion detection, mobile security, secure systems, and network design.

We sincerely thank all those who submitted papers as well as the Program Committee members and external reviewers for their valuable contributions to an excellent conference program.

For further details about DIMVA 2012, please refer to the conference website at http://www.dimva.org/dimva2012.

July 2012

Ulrich Flegel
Evangelos Markatos
William Robertson

# Organization

DIMVA 2012 was organized by the Special Interest Group Security – Intrusion Detection and Response (SIDAR) of the German Informatics Society (GI).

## Organizing Committee

| | |
|---|---|
| General Chair | Evangelos Markatos (FORTH) |
| Program Chair | William Robertson (Northeastern University) |
| Financial Chair | Ulrich Flegel (HFT Stuttgart) |

## Program Committee

| | |
|---|---|
| Davide Balzarotti | Eurécom, France |
| Erik-Oliver Blass | Northeastern University |
| Juan Caballero | IMDEA-Software |
| Lorenzo Cavallaro | Royal Holloway, University of London, UK |
| Stephen Checkoway | UC San Diego, USA |
| Mihai Christodorescu | IBM Research |
| Marco Cova | Birmingham University, UK |
| Manuel Egele | UC Santa Barbara, USA |
| Ulrich Flegel | HFT Stuttgart University of Applied Sciences, Germany |
| Felix Freiling | Friedrich Alexander University, Germany |
| Chris Grier | ICSI, UC Berkeley, USA |
| Guofei Gu | Texas A&M |
| Thorsten Holz | Ruhr University Bochum, Germany |
| Martin Johns | Universität Passau, Germany |
| Andrea Lanzi | Eurécom, France |
| Wenke Lee | Georgia Tech, USA |
| Corrado Leita | Symantec Research Labs |
| Ben Livshits | Microsoft Research |
| Lorenzo Martignoni | Google |
| Michael Meier | University of Bonn, Germany |
| Paolo Milani Comparetti | LastLine |
| Collin Mulliner | TU Berlin, Germany |
| Roberto Perdisci | University of Georgia, USA |
| Adrienne Porter Felt | UC Berkeley, USA |
| Konrad Rieck | University of Göttingen, Germany |
| Giovanni Vigna | UC Santa Barbara, USA |
| Heng Yin | Syracuse University, USA |

## Steering Committee

### Chairs

| | |
|---|---|
| Ulrich Flegel | HFT Stuttgart, Germany |
| Michael Meier | University of Bonn, Germany |

### Members

| | |
|---|---|
| Herbert Bos | VU University Amsterdam, The Netherlands |
| Danilo M. Bruschi | Università degli Studi di Milano, Italy |
| Roland Büschkes | RWE AG, Germany |
| Hervé Debar | Télécom SudParis, France |
| Bernhard Haemmerli | Acris GmbH & HSLU Lucerne, Switzerland |
| Marc Heuse | Baseline Security Consulting, Germany |
| Thorsten Holz | Ruhr-University Bochum, Germany |
| Marko Jahnke | Fraunhofer FKIE, Germany |
| Klaus Julisch | Deloitte, Switzerland |
| Christian Kreibich | International Computer Science Institute, USA |
| Christopher Kruegel | UC Santa Barbara, USA |
| Pavel Laskov | University of Tübingen, Germany |
| Robin Sommer | ICSI/LBNL, USA |
| Diego Zamboni | CFEngine AS, Norway |

# Table of Contents

## IDS

# Using File Relationships in Malware Classification

Nikos Karampatziakis[1], Jack W. Stokes[2], Anil Thomas[1], and Mady Marinescu[1]

[1] Microsoft Corporation, One Microsoft Way, Redmond WA, 98052, USA
just.nikos@live.com, {anilth,mady}@microsoft.com
[2] Microsoft Research, One Microsoft Way, Redmond WA, 98052, USA
jstokes@microsoft.com

**Abstract.** Typical malware classification methods analyze unknown files in isolation. However, this ignores valuable relationships between malware files, such as containment in a zip archive, dropping, or downloading. We present a new malware classification system based on a graph induced by file relationships, and, as a proof of concept, analyze containment relationships, for which we have much available data. However our methodology is general, relying only on an initial estimate for some of the files in our data and on propagating information along the edges of the graph. It can thus be applied to other types of file relationships. We show that since malicious files are often included in multiple malware containers, the system's detection accuracy can be significantly improved, particularly at low false positive rates which are the main operating points for automated malware classifiers. For example at a false positive rate of 0.2%, the false negative rate decreases from 42.1% to 15.2%. Finally, the new system is highly scalable; our basic implementation can learn good classifiers from a large, bipartite graph including over 719 thousand containers and 3.4 million files in a total of 16 minutes.

**Keywords:** Malware detection, Machine Learning, File Relationships.

## 1 Introduction

Symantec recently observed over 903 million files installed on a sample of 47 million computers [3]. While many of these single instance files are benign, a significant percentage are malicious. Malicious single instance files have two sources, polymorphic and metamorphic malware which installs a unique instance of the attack on each new computer, while legitimate software sometimes creates a unique file for each installation. Given the shear volume, human analysts cannot investigate each new file detected in the wild, and the anti-malware (AM) companies cannot solve the problem by hiring more analysts. Since malware authors often rely on automation to avoid detection, commercial anti-malware companies also need to use automation to detect new malware. Ideally, the most automatic and effective way to solve the problem would be to have secure systems that would not allow the execution of malicious code. However this paradigm is

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 1–20, 2013.

currently far from realistic while malware proliferation is already a sad reality. Since malware is such a significant problem, researchers and industry have devoted much effort towards automated detection [9]. The main issue with these systems is that the false positive (FP) rates (i.e. when a benign file is predicted as malicious) are too high for widespread deployment. An acceptable FP rate for completely automated malware detection would be no more than 0.01%. For AM systems, FPs are much worse than false negatives (FNs) since cleaning (i.e. removing) FPs can prevent a legitimate application, or even the operating system, from running. Given this risk, most AM companies seek first to do no harm.

Current techniques for building malware classifiers lead to prohibitively high FN rates (e.g. $> 99\%$) when operating at an FP rate of 0.01%. To truly combat today's malware, we need to look from new sources of information. For example, a system that uses both the static structure of a file as well as information about its runtime behavior is likely to perform much better than systems that use only one of these sources. In this paper we use a different source of information, namely file relationships, and demonstrate that even a very simple way of incorporating this source into the classifier is very effective.

Typical approaches [12,20] focus on classifying files in *isolation*. Recently, researchers have proposed using a file's reputation [3] in relationship to the reputations of all computers which report the file to improve the detection rate. Other authors [22] rely on co-occurrences of files on a set of client machines to establish threats such as trojan downloaders. In this work, we take a different approach to improving a malware classifier by learning a file's reputation based on its relationships with other files as we determine them through Microsoft's submission service. We empirically show improved classification accuracy at very low FP rates. Furthermore, we sidestep privacy issues that affect other approaches.

File relationships should contain useful information. Clearly it is enough to show this for a special case. In this paper we only consider containment relationships, the most prevalent type in our data. However, our algorithm is quite general and could be applied to other types of file relationships. Containment arises when malware is distributed in containers such as .zip or .rar files. We also ignore containment relationships among containers (archives containing archives) and only form a bipartite graph of files and containers. After these restrictions, our database has more than four million containers with more than one executable file inside them and exploiting this information may substantially improve over a classifier that ignores it.

Our method starts by training a *baseline classifier* to *individually* predict the probability that a file is malicious. This classifier is trained with over 2.6 million labeled files using logistic regression. This classifier, though simple, uses some very strong features coming from an actual execution of the file in a virtual machine, and is very fast during training and prediction. Our method then propagates this baseline prediction and other information from files to containers using the file relationship graph. It then trains a *container classifier* which assigns a probability to each archive. Here we investigate three ways of integrating information from the neighboring vertices. Finally, we significantly improve

the classification accuracy on individual files by training a *relationship classifier*, again using logistic regression, which considers the malware probabilities associated with all archives containing the file in addition to the file's baseline score. Experiments on a large collection of over 719 thousand archives including more than 3.4 million malicious and benign files show that the relationship classifier significantly outperforms the baseline classifier particularly at very low FP rates. For example, at an FP rate of 0.2%, the FN rate decreases from 42.1% to 15.2%.

In Section 2, we provide some background on using machine learning for malware classification. Our new container classifier and file relationship classifier are described in detail in Section 3. An overview of the system is presented in Section 4, and experimental results are given in Section 5. In Section 6, we discuss the assumptions of our method, and the related work is outlined in Section 7. Finally, we conclude the paper in Section 8. Our main contributions include:

- A large-scale system to classify unknown files based on file relationships.
- A comparison of three methods to classify file containers.
- A new algorithm for significantly improving a file's baseline prediction.
- An evaluation of our system on a large collection of over 719 thousand containers and 3.4 million files, where we demonstrate that it is highly scalable.
- A convincing empirical comparison based on performance at small FP rates.

## 2 Background and Notation

Most modern anti-malware software follows a rule-based method to detect malware usually referred to as "signatures". Recently, researchers [9] have proposed using machine learning *classifiers* to detect malware. Classifiers built from a set of labeled malicious and benign programs can generalize well to previously unseen but similar programs. Here we focus on linear logistic classifiers because:

- They allow our experimental results to be directly interpretable when the classifier is employed in the real world. In the real world we do not know the proportion of actual malware files. Remarkably, as we show later in the paper, the conclusions we draw from logistic regression remain valid.
- Our system builds on top of a baseline classifier that works on individual files. This classifier is itself based on logistic regression and we exploit this fact to provide an initial hint to our system.
- Finally a logistic classifier can make predictions very fast, which is highly desirable for the scalability of our system.

For each file $x_i$ we would like to classify, we construct a vector of real numbers $\Phi(x_i) \in \mathbb{R}^d$ that contains measurements, also known as *features*, regarding the file $x_i$. Described in Section 4.2, one of the features we use for our baseline classifier consists of tri-grams of system calls. In general we assume that the mapping $\Phi(\cdot)$, that takes an executable and returns a vector in $\mathbb{R}^d$, is given to us. In typical cases, to get a descriptive enough representation of $x_i$, $d$ is on the order of hundreds of thousands but for each individual $x_i$, $\Phi(x_i)$ is sparse, meaning that only few (on the order of a hundreds or a few thousand) of the entries in this vector are non-zero.

### 2.1 Classification with Logistic Regression

Logistic regression assumes that for each file $x_i$ we can assign a score $s(x_i)$ that is a linear combination of the feature values for $x_i$, or more formally $s(x_i) = \sum_{j=0}^{d} w_j \cdot \Phi_j(x_i) = w^\top \Phi(x_i)$ for some, yet to be determined, *weights* $w$ where $\Phi_0(x_i) = 1$, $w_0$ is the bias term, and $a^\top b$ denotes the inner product between vectors $a$ and $b$. This score is converted into a probability using the *logistic link function* $g(t) = \frac{e^t}{1+e^t}$. Letting $y_i$ be a random variable that is 1 if the file $x_i$ is malware, and 0 otherwise, logistic regression assumes

$$p(y_i = 1|x_i) = g(s(x_i)) = \frac{e^{w^\top \Phi(x_i)}}{1 + e^{w^\top \Phi(x_i)}}. \tag{1}$$

Solving (1) for the score $s(x_i)$ we find that

$$s(x_i) = w^\top \Phi(x_i) = \log \frac{p(y_i = 1|x_i)}{1 - p(y_i = 1|x_i)} = \log \frac{p(y_i = 1|x_i)}{p(y_i = 0|x_i)}.$$

The right hand side is commonly referred to as the *log odds* of $x_i$ being malware. We will return to this relation in Sections 3.5 and 3.6. Finally, finding the optimal weight vector $w$ from a training set of files for which human analysts have provided *determinations* (i.e. whether the file is truly malware or not), is a well studied convex optimization problem for which extremely fast algorithms exist.

## 3 Beyond Individual Prediction

In this section, we develop new methods whose goal is to improve malware classification particularly at low FP rates. We begin by discussing two problems associated with typical classification algorithms and then propose using a file's relationships to overcome these issues. After considering an ideal, but impractical solution, we propose a new set of algorithms to achieve a similar effect.

### 3.1 Some Problems of the Standard Approach

First we argue that current approaches to malware classification have a lot of room for improvement if we consider how an analyst would go about determining whether a file is malware or not. Malicious files do not exist in a vacuum. Malware is often distributed in an archive and this reflects a relationship among these files. The exact meaning of the relationship varies from archive to archive. Some typical examples include an executable file and its dependencies (such as dynamically linked libraries), files created by the same author, or components of a large project. In any case, this is precious information that is not captured in the framework of individually predicting on each file, but is routinely leveraged by malware analysts. That is, approaches based on individual prediction ignore the relationships of the file under consideration to other related files and their determinations. This is not just a matter of extending the feature mapping $\Phi(\cdot)$ to include features from the related files. A related file itself may not reveal anything alarming but it may do so if one considers its own related files. Later in Section 5.1, we motivate this idea based on two particular examples.

### 3.2 Taking Context into Account

We propose to overcome the shortcomings of individual malware classification with a two step procedure which we describe in Sections 3.5 and 3.6. Before that, we introduce some terminology, and describe an ideal but impractical approach. We represent the file relationships with a graph in which the vertices correspond to files and the edges correspond to containment relationships. Even though containers can contain other containers, here we ignore this and focus on a bipartite graph between containers and regular files. We have an additional determination "malware container" which is given to containers that contain at least one file determined to be "malware". Of course, not all vertices have a determination of "malware" or "benign", and we would like to propagate information along the graph so that we can assign each vertex its own probability that it is malicious.

We immediately point out that such context information cannot be easily made available to anti-malware software running on a client, and facilitating this functionality to a client is beyond the scope of this paper. Our focus is to demonstrate the empirical gains we observe when this information is available and utilized on the backend with an approach that is relatively easy to implement and highly scalable. The method we advocate however, is inspired by an impractical solution. We nevertheless detail this impractical solution in the next section to explicate the ideas that lead to our scalable algorithm.

### 3.3 An Impractical Solution

To assign a malware probability to every file we assume we have a *baseline malware classifier* that employs the approach of Section 2 to assign a score $w^\top \Phi(x_i)$, and hence a probability via the logistic link function, to each executable file in our data. For the files already determined to be malware or benign we can define the maliciousness level to be the score from the baseline malware classifier. For all other files for which we have no determination we can formulate a set of *consistency equations* according to a very simple principle: *we can obtain the maliciousness level of a file by averaging the maliciousness levels, of its related files*, which of course are its neighbors in the graph. This definition treats maliciousness as a fixed point. Formally, letting $s_i$ denote the maliciousness level for file $i$ and $N(i)$ the set of neighbors of $i$ in the graph we have

$$s_i = \begin{cases} w^\top \Phi(x_i) & \text{if } i \text{ is determined} \\ \frac{1}{|N(i)|} \sum_{j \in N(i)} s_j & \text{otherwise} \end{cases}$$

For $m$ files, this defines an $m \times m$ system of linear equations. Typically $m$ is huge; in our case $m$ is greater than 250 million and is growing by two every second. This immediately precludes methods such as Gauss elimination [7] which scale as $O(m^3)$. Furthermore, the graph, and hence each equation, is also evolving because each submission to our service can induce new relationships. Hence the solution of the above linear system is largely of theoretical interest even if one seeks an approximate solution using iterative methods [7].

### 3.4 A Scalable Solution

The main problem with the previous solution is that it strives to be globally consistent and enforcing this is too time-consuming. Instead we relax the requirement for consistency and, to compensate for this, we make our aggregation rule more flexible. Instead of a simple average, we compute features of the immediate neighborhood of each file and find an optimal way to combine them. Specifically, our solution involves the following steps:

- compute a "malware probability" for each container by aggregating information from all the files it contains and the probabilities assigned to them by our baseline malware classifier.
- compute an improved estimate of the probability of a file being malware by aggregating malware probabilities from the archives that contain it as well as the baseline malware classifier.

### 3.5 Computing Container Probabilities

A malware analyst does not have to look at the whole graph to get a rough idea about how likely the file is to be malicious; the local neighborhood provides most of the information. Furthermore, we observe that *in order for a container to be malicious it suffices that one of its contained files is malware.* Conversely, a container has a low malware probability only when all the contained files are not malicious. Based on this observation we propose three "container classifier" algorithms for assigning a malware probability to each container: Max neighbor, Union bound, and Biased logistic regression.

The Max neighbor algorithm estimates the probability that an archive is a "malware container" by the maximum of the probabilities (as given by the baseline classifier) of any of the contained files being malware. The Union bound makes a simple assumption: each file is independently providing evidence about the maliciousness of the container. Hence, the probability of the container to be benign is the product over all contained files of their probabilities of being benign. For example, if an archive contains two files, one for which the baseline estimate that it is malware is 0.6 and another whose estimate is 0.5 then we assign a probability of $1 - (1 - 0.6)(1 - 0.5) = 0.8$ to the container being malware.

Biased logistic regression employs a logistic regression classifier with an additional offset motivated by the importance of the file with the maximum baseline probability contained in the archive:

$$\log \frac{p_c(y_i = 1|N(i))}{p_c(y_i = 0|N(i))} = v^\top \Psi(N(i)) + v' \log \frac{p_{b,\max}}{1 - p_{b,\max}} \tag{2}$$

where $N(i)$ is the set of files contained in archive $i$, $v$ (vector) and $v'$ are the model weights, $p_{b,\max}$ is the maximum of the probabilities assigned to all the files in $N(i)$ by the baseline malware classifier, and $\Psi(N(i))$ is a vector of features calculated from the files contained in archive $i$. This model is biasing its prediction based on the most malicious file among $i$'s files, in accordance to our observation that a container is malicious if at least one of the contained files

is malicious. Furthermore we adjust this prior belief by a linear combination of features, $v^\top \Psi(N(i))$, computed from the neighborhood of $i$ which captures the *maliciousness levels* of all of the files in the container. Most of these features come from three simple histograms of the baseline probability estimates of the files included in the container. The histograms are separately computed for files which are predicted to be malicious or benign by the baseline classifier. A third histogram is included for files for which the baseline classifier returned a probability but a label of inconclusive. For each type, we split the interval $[0, 1]$ into 20 equally sized bins and create a histogram of the probabilities of the contained files. Then the values of features $\Psi_{2j}$ and $\Psi_{2j+1}$ are respectively the fraction and the logarithm of the number of contained files whose baseline probability estimates fall in the $j$-th bin. We chose these features to capture both absolute and relative numbers that may affect our decision. These transformations are relatively insensitive to manipulation of the raw numbers from adversaries. We also include three additional features for the container classifier. The first is the biasing feature which is the inverse of the logistic link function. Since the baseline malware classifier is itself based on logistic regression, the biasing feature is $\log(p_{b,\max}/(1 - p_{b,\max}))$ where $p_{b,\max}$ is the contained file with the highest probability. As $(p_{b,\max})$ approaches 1, the biasing term becomes large and hence overshadows any effect from $v^\top \Psi(N(i))$. When $p_{b,\max}$ approaches 0, the biasing term becomes very negative and again overshadows the effect of $v^\top \Psi(N(i))$. Finally when the max probability is 0.5 the biasing term is 0. The second additional feature is the log of the number of files in the container and the third is the product of the first two additional features. The third additional feature captures interactions between the number of files and the maximum file probability in the container. The last two features were important for reducing the number of false negatives for the container classifier. The entire container classifier procedure is summarized in the top part of Table 1.

The main benefit of this approach is that it is extremely fast to make a prediction for a new container. We only need to look at its contained files, and retrieve their probabilities from our database. If a file has not been seen before, we need to obtain its probability from the baseline malware classifier, compute 123 features from three histogram (3*20 bins and two features from each bin plus the three additional features), and take a linear combination with the learned vector $v$.

### 3.6 Improving a File's Probability

Our end goal is to improve upon a system that classifies executable files individually. In this sense, the probabilities we obtain from the container classifier is just auxiliary information that summarizes the neighborhood of a file. Therefore we introduce a second step where *we aggregate information across the containers in which a given file participates*. Our final "relationship-based" classifier uses a similar set of features as the container-based classifier as well as a biasing term. However this time our prior belief reflected in the biasing term is that the baseline classifier is doing well most of the time, and we only want to use the

**Table 1.** Algorithm for Improving File Malware Probabilities

| Algorithm 1 |
|---|
| Notation: $g(z) = \frac{e^z}{1+e^z}$, $\quad t(w, \phi, q) = w^\top \phi + \log \frac{q}{1-q}$. |
| Let $C$ be the set of labeled containers |
| **Collect Container Data**: Let $S_c = \{(\Psi(N(i)), p_i, y_i) \mid i \in C\}$ where $N(i)$ is the set of executables in a container $i$, and $\Psi(\cdot)$ is computed from $p_b(y_j = 1 \mid x_j)$, $j \in N(i)$ $p_i = \max_{j \in N(i)} p_b(y_j = 1 \mid x_j)$. |
| **Train Container Classifier**: Find the vector $v^*$ that maximizes $L_c(v) = \prod_{(\psi,p,y) \in S_c} g(t(v, \psi, p))^{y_i} (1 - g(t(v, \psi, p)))^{1-y_i}$ |
| **Assign Container Probabilities**: For each container $i$: $p_c(y_i = 1 \mid x_i) = g(t(v^*, \Psi(N(i)), p_i))$ |
| Let $F$ be the set of labeled files. |
| **Collect File Data**: Let $S_f = \{(\Psi(N(i)), p_b(y_i = 1 \mid x_i), y_i) \mid i \in F\}$ where $N(i)$ is the set of containers containing $i$ and and $\Psi(\cdot)$ is computed from $p_c(y_j = 1 \mid x_j)$, $j \in N(i)$. |
| **Train Relationship Classifier**: Find the vector $u^*$ that maximizes $L_r(u) = \prod_{(\psi,p,y) \in S_f} g(t(u, \psi, p))^{y_i} (1 - g(t(u, \psi, p))^{1-y_i}$ |
| **Improve File Probabilities**: For each file $i$: $p_r(y_i = 1 \mid x_i) = g(t(u^*, \Psi(N(i)), p_b(y_i = 1 \mid x_i)))$. |

neighborhood information to learn a *correction*. Formally, we model the log odds of file $i$ being malware ($y_i = 1$) as

$$\log \frac{p_r(y_i = 1 \mid N(i))}{p_r(y_i = 0 \mid N(i))} = u^\top \Psi(N(i)) + u' \log \frac{p_b(y_i = 1 \mid x_i)}{p_b(y_i = 0 \mid x_i)} \tag{3}$$

where $p_r$ is the probability according to the relationship-based classifier, $p_b$ is the probability according to the baseline classifier, $\Psi$ is a vector of features we compute from the neighborhood of the file and $u$ is a vector of weights that optimally combines the features according to the maximum likelihood principle and $u'$ is a weight for biasing term. The term $u^\top \Psi(N(i))$ captures the maliciousness level of all of the containers which include the file under consideration. This time we derive the mapping to features $\Psi$ by binning the probability estimates from the container classifier into two histograms (malware, benign) for each of the neighboring containers. If a neighbor is a new container and has not been classified yet we simply ignore it. This is fine since, as before, the neighborhood features will come to the rescue mostly when the baseline classifier's prediction is close to 0.5 and will be overshadowed by the biasing term as the baseline classifier becomes very confident. An additional feature of $\log \frac{p_b(y_i=1 \mid x_i)}{p_b(y_i=0 \mid x_i)}$ helps to bias the model to the baseline probability. The whole procedure is shown in Table 1. Our non-optimized implementation executes it in 16 minutes processing 719 thousand containers and 3.4 million files.

One could argue at this point that we could go back and, based on the improved probability estimates from the relationship classifier, compute new probabilities for archives. We could in fact iterate this procedure until it converges to a fixed point. However it is doubtful that such a fixed point will lead to substantially better *generalization* than our procedure. First, the fixed point integrates information from many potentially unrelated files which are simply too far from the file under consideration. Second, it is well known among machine learning practitioners that treating the output of one classifier as an input to another (aka cascading) is an extremely delicate procedure. Hence it is usually observed that, as a function of the length of the cascade, the generalization performance of the final output rapidly deteriorates.

## 4 System

This section discusses the system aspects related to training the relationship malware classifier illustrated in Figure 1. The data analyzed in this paper consists of a very large subsample of containers and files used by Microsoft to investigate and create signatures for a number of our commercial anti-virus products including Microsoft Security Essentials and Forefront Endpoint Protection. Microsoft collects suspicious files using a variety of sources including the end user, product support, security organizations (e.g. CERT), and vendor exchange. After submission, each unknown file is automatically scanned by our AM products. Some files are detected by the scanners, and a small subset of the undetected files are investigated manually by analysts for additional signature creation. In addition, we have a large collection of programs known to be legitimate (i.e. clean); many of these programs include one or more containers. Labels (i.e. "malware container", "benign container", "malware", "benign") are assigned to the containers and individual files depending on the source. Some of these labeled files are then used to train the baseline malware classifier. As part of the scanning process, container files (e.g. .zip, .rar) are uncompressed, and the individual files are extracted. A graph is constructed based on the relationships observed in the containers, and in parallel, the trained baseline classifier is used to predict the probability that each individual file is malicious. In another part of the scanning process, a file is run and any files which are dropped (i.e. written to the disk drive) are detected. These "dropped" relationships could also be used in our system. The datasets used to train the baseline and relationship classifiers differ because while all of the individual files used to train the baseline classifier have previously been labeled, most of the files in the containers have not. In the last step, the relationship malware classifier is trained using the baseline file predictions and the relationships from the graph. In the remainder of this section, we further investigate the container details and describe the baseline classifier.

### 4.1 Container Description

To train the classifier, we first obtain labels for the containers by assigning the label "malware container" to containers which were either previously determined

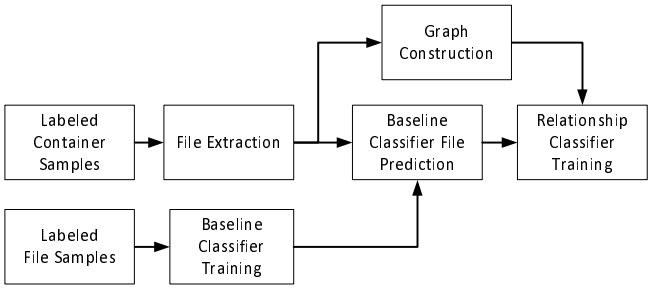



**Fig. 1.** System Diagram for Training the Relationship Classifier

as malicious by an analyst, one or more contained files were labeled as malware by an analyst, or an anti-virus engine automatically detected at least one file as malware in the container. Similarly, benign archives are labeled as "benign container" and defined as those previously determined as benign by an analyst, contain no files that were labeled as malware by an analyst or other automated system, and an anti-virus engine did not automatically detect any of its contained files as malware. Next, we constructed a labeled graph containing 4,160,807 nodes and 23,993,309 edges where each node is either an archive or individual file and an edge indicates that an archive contains a file. This graph includes 719,359 total archives including 604,658 malicious and 114,701 benign containers. There are 3,441,448 individual files with 492,443 labeled as malicious and 2,949,005 labeled as benign. Among the individual files, 67,705 of them existed both in malware and in benign containers.

Figures 2 and 3 present the distributions of the number of files included in the malware and clean containers, respectively. The approximately linear relationship of the files in the malware containers on a log-log scale indicates that the number of files in the malware containers roughly follows a power law. On the other hand, many of the benign containers are distinct versions of commercially available software products including multiple versions of the same product written in many different languages. These programs often contain similar, but distinctly different, numbers of files leading to multiple archives with approximately the same number of files. This behavior is also noted for multiple versions of the same program (e.g. Adobe Acrobat Reader versions 7.0 and 7.1).

In Figures 4 and 5 we show the distribution of the number of archives that include each malware and clean file respectively. Again we observe a power law behavior for the malware, with most malware files appearing in very few containers and only a handful of malware files appearing in many containers. On the other hand for the benign files the power law behavior does not span as many orders of magnitude and instead we observe that there are several benign files that are contained in many archives. The reason for this discrepancy is that benign software is made with the intent to be reused while malware is created for more opportunistic purposes.

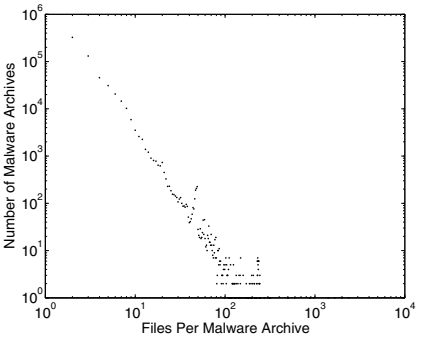



**Fig. 2.** Distribution of Files in the Malware Containers

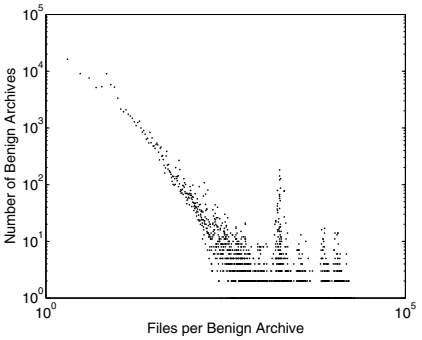



**Fig. 3.** Distribution of Files in the Benign Containers

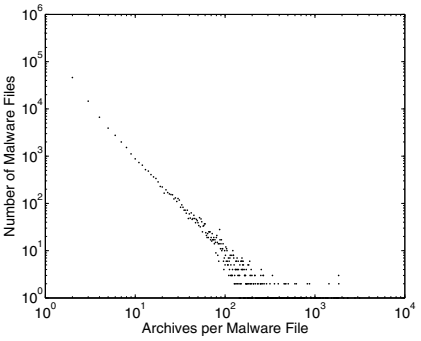



**Fig. 4.** Distribution of Archives which Contain Malware Files

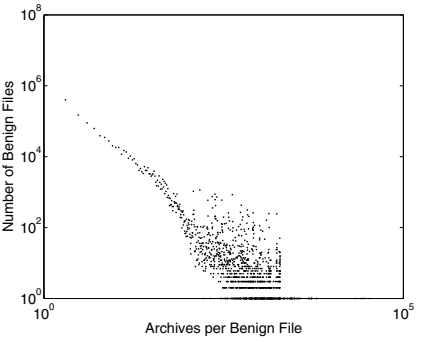



**Fig. 5.** Distribution of Archives which Contain Benign Files

### 4.2 Baseline Classifier

In this section, we briefly describe the baseline malware classifier which provides the original probability that an unknown file is malicious or benign. We first collected over 2.6 million samples consisting of 1,843,359 malware files and 817,485 samples of files known to be benign. Each of the malicious files was also assigned to a particular malware family. We selected a set of 134 malware families determined by analysts to be important to identify. All malware samples belonging to malware families not in the set were included in a generic malware class, and all samples of legitimate software were assigned to a benign class. Next, we modified the production anti-malware engine used in Microsoft's commercial security products (e.g. Microsoft Security Essentials, Forefront Endpoint Protection) as well as Windows (i.e. Windows Defender) to produce a set of log files for further analysis. As part of the overall analysis, this AM engine runs each unknown file

in a lightweight virtual machine. We then record each system call and its corresponding input parameters. We also extracted the process memory and searched for null-terminated patterns. This collection of patterns often includes strings but sometimes include snippets of code. From these logs, we created four sets of potential features for the baseline classifier. First, we identified each distinct combination of a system call and its parameter values. For example, if the unknown executable calls CreateThread() with a stack size of 1 megabyte, this API/parameter combination would then serve as one potential feature. Next, we constructed tri-grams of the system call sequence. We also included each of the process memory patterns in the pool of potential features. Finally, we included 164 low-fidelity features from analysis of the file such as: is it 64-bit software?, is it an .EXE file?, is it a .DLL file?, and so on. In practice, this last set of features was overwhelmed by the other feature sets and only improved the model's accuracy by 0.01%. It should be noted that our features are constrained by the limitations of the production anti-malware engine. As a result, we cannot use features which require significant time to compute such as the system call graph using dynamic taint analysis. Processing the logs produced a set of over 50 million potential features which needs to be significantly reduced to avoid overfitting. Based on mutual information [13], we then used feature selection of the potential feature set to determine 179,288 features for the classifier. Finally with these selected features, we constructed a labeled training set to train a multi-class classifier with logistic regression using stochastic gradient descent (SGD) to predict if an unknown file belongs to one of the malware families under consideration or to the generic malware or benign class. We chose to train a logistic regression model with SGD because of the large scale nature of our data in terms of both the number of samples and features. Due to our implementation in C# and a .NET memory constraint related to the number of elements in a list, we used SGD to efficiently train the baseline classifier in mini-batches of roughly 450 thousand examples. Training the multi-class classifier produced a false positive of 1.3% and a false negative rate of 0.7% on a separate (i.e. hold-out) test set of over 443 thousand files. Even though the features used in this classifier are relatively simple, training with over 2.6 million files produces a good error rate. Furthermore, malware classification research has produced a number of independent algorithmic improvements to increase malware classification accuracies [18,17,15,2,11,16,6]. As we argue in Section 8, since the baseline classifier and container relationship structure are independent, applying one or more of these algorithmic improvements or additional feature sets to the baseline classifier will help to improve the overall system response.

## 5 Experimental Results

In this section, we conduct a series of experiments to evaluate the performance of the container and improved relationship malware classifiers.

**Table 2.** An Example of the File Containers which Include `2.exe`

| Name | Determination | # Scanner Detections | # Submissions |
|---|---|---|---|
| . . . _Norton _Antivirus . . . _2007 _.rar | Malware Container | 15 | 2 |
| . . . ba52.bin | Malware Container | 15 | 4 |
| . . . _z0ffzvk _.rar.part | Malware Container | 14 | 2 |
| . . . _dc11.rar | Malware Container | 14 | 2 |
| . . . _regcure _1.0.0.43.1.3a1400.efw | Malware Container | 14 | 2 |
| . . . _Registry _Mechanic . . . .rar | Malware Container | 14 | 2 |
| . . . _CyberLink _PowerDVD _7.0.rar | Malware Container | 15 | 2 |
| . . . _adobe _photoshop _cs2 _.rar | Malware Container | 15 | 2 |

### 5.1 Examples

We first motivate the relationship classifier idea by examining how it affects the baseline probability of two individual files. Upon manual examination of the first file `2.exe`, we found this file to be a variant of a Trojan in the Vundo family. Table 2 indicates the file was included in 8 containers, which were labeled "Malware Container". This table includes the names of the containers, their determinations, the number of scanners which detect them, and the number of submissions. The third column indicates all containers were detected by at least 14 scanners. For this example, the baseline malware classifier failed to correctly identify the file as malicious. The relationship classifier raised the probability of this file from a baseline of 33% to 98.37% which is more indicative of malware. This shows that the malware relationship classifier can help to correctly identify malicious files even when the baseline classifier misclassifies them.

The second example involves a file named `calleng.dll`. The file was manually determined to be benign by an analyst, and the baseline malware classifier assigns a probability of nearly 0% that this file is malware. We scanned it with a set of anti-malware scanners and no scanners detected the file. This file was originally distributed as part of the legitimate `PalTalk` social networking software. Table 3 provides the container relationships. We believe that `(RarSfx)` on row 4 with no detections is the legitimate `PalTalk`. We also have evidence from the scanner detection column that the remainder of the containers in Table 3 are indeed suspicious. In fact, we believe that these are malicious versions of the original `PalTalk` application. While `calleng.dll` itself is not malicious, it clearly appears to be commonly used by malware authors in some manner. In other words, a previously unseen archive containing this file is likely to be malware. After running the malware relationship classifier on `calleng.dll` the malware probability increased to 44.9%. While this is certainly more indicative of being malicious, it is not sufficient to be classified as malware. This shows that even when the new probability estimates of a benign file are increased, this is usually not enough to cause a false positive. This is confirmed by the overall improved results in Figure 7.

**Table 3.** An Example of the File Containers which Include `calleng.dll`

| Name | Determination | # Scanner Detections | # Submissions |
|---|---|---|---|
| 0d... bc.rar | No Determination | 13 | 2 |
| d3... 39.rar | No Determination | 9 | 2 |
| ec... da | No Determination | 3 | 2 |
| (RarSfx) | No Determination | 0 | 2 |
| (RarSfx) | No Determination | 7 | 4 |
| (RarSfx) | No Determination | 9 | 4 |

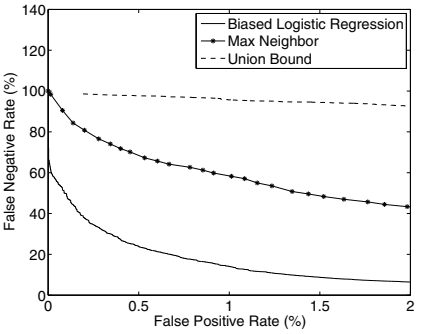



**Fig. 6.** DET Curves for the Container Classifier Algorithms

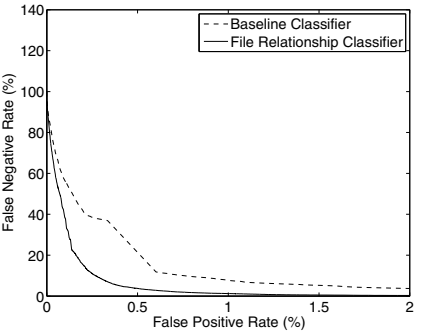



**Fig. 7.** DET Curves for the Baseline and Relationship File Classifiers

### 5.2 Performance Analysis

Detection Error Tradeoff (DET) curves for the three container classification algorithms proposed in Section 3.5 are plotted in Figure 6. To obtain probabilities for all containers with the container classifier in a fair way we used 5-fold cross validation. If the DET curve of a rule is always below the DET curve of another rule then we say that the former *dominates* the latter. It means that for all possible FP rates one classifier is always achieving better FN rates than another; a very strong statement. For malware detection, we care about the region of small FP rates, and in the figures we are zooming in a range of up to 2% FP rate. In this range, the Biased logistic regression algorithm completely dominates the simple approaches of the Max neighbor and Union bound algorithms. In fact, the Union bound method is dominated by the Max neighbor rule which demonstrates the inappropriateness of its assumptions. This leads us to believe that the files in the containers are correlated and reinforces our belief that aggregating information across the files in the container is not trivial. We mention that the Biased logistic regression method dominates the other two rules across all the FP rates, not just across the range shown in Figure 6.

In Figure 7 we present DET curves for individual files. We compare the existing baseline classifier with the relationship classifier of Section 3.6, and we again employ 5-fold cross-validation. As before, for malware classification we are

**Table 4.** Comparison of Baseline and File Relationship Classifier Statistics for FP Rate = 1% and 0.1%

| FP Rate | Label | $p_b \leq t_b$ $p_r \leq t_r$ | $p_b > t_b$ $p_r \leq t_r$ | $p_b \leq t_b$ $p_r > t_r$ | $p_b > t_b$ $p_r > t_r$ |
|---|---|---|---|---|---|
| 1.0% | Malware | 6269 | 161 | 32170 | 480548 |
| | Benign | 2909583 | 15561 | 14590 | 14959 |
| 0.1% | Malware | 183454 | 15406 | 109043 | 211245 |
| | Benign | 2950180 | 1556 | 1546 | 1411 |

interested in small FP rates and therefore plot the curves for FP rates up to 2%. We observe that the relationship classifier convincingly dominates our baseline system: At an FP rate of 0.3%, there is a 77.3% decrease in FN rate (from 37.5% to 8.5%). At our target FP rate of 0.01%, the decrease in the FN rate from 87% to 85% is marginal; there is still more work to be done to achieve widespread detection at these extremely low FP rates. In the rest of the FP range (not shown), the baseline system remains dominated until approximately an FP rate of 60%, when it becomes marginally advantageous to use the baseline system. However, these operating points are uninteresting even for perimeter-based anti-malware systems. Our conclusion is that our approach not only improves upon the baseline system, but it does so for error tradeoffs that are important for our application. In Section 5.3 we explain why (and this is true only for logistic regression) the results of Figures 6 and 7 are invariant to the proportion of malware files we used in the experiments.

In Table 4 we next compare example counts for the baseline and relationship classifiers for different threshold values corresponding to FP rates of 1% and 0.1% . For the row labeled malware in Table 4 with a FP rate of 1%, the sixth ($p_b > t_r, p_r > t_r$) and third ($p_b \leq t_b, p_r \leq t_r$) columns indicate that both classifiers correctly identify 480,548 malicious files but fail to detect 6,269 files. The fourth column indicates that 161 files were incorrectly mispredicted by the relationship classifier as benign (FN) but correctly predicted by the baseline system. The fifth column shows the improvement of the relationship classifier for 32,170 files which were mispredicted by the baseline classifier. For the second row with benign files at 1% FP rate, 15,561 files were correctly predicted to be benign by the relationship classifier but missed by the baseline classifier. Likewise at an FP rate of 1%, the relationship classifier had 14,590 false positives. Similar statistics are noted for a 0.1% FP rate. Note that the baseline classifier threshold ($t_b$) and relationship classifier threshold ($t_r$) differ in order to set the operating point to the desired FP rate. Also, the number of FPs for the baseline classifier ($p_b > t_r, p_r \leq t_r$) and for the relationship classifier ($p_b \leq t_b, p_r > t_r$) are approximately equal, but differ slightly due to multiple examples having identical predicted probabilities.

In Figures 8 and 9, we further investigate the 14,590 FPs generated by the relationship classifier at a FP rate of 1%. Figure 8 shows a histogram of the FPs that are found in more than 10 containers. This figure accounts for approximately two thirds of our FPs. The FPs have been binned according to the fraction of

associated containers that are malware containers. We see that a large number of our FPs are associated with containers most of which contain malware. In fact, 31.5% of our FPs are files found in containers in which the majority of files are malware. Figure 9 shows a heatmap for the rest of the FPs, i.e. those found in 10 or less containers. Here we see that the overwhelming majority are associated with only one container which contains no malware.

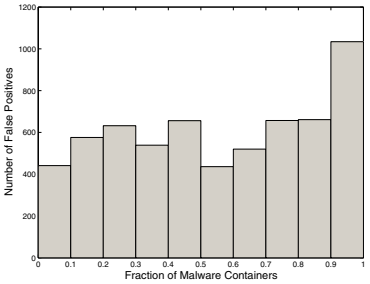



**Fig. 8.** FPs Binned by the Fraction of Malware Containers Out of All Archives Containing Each

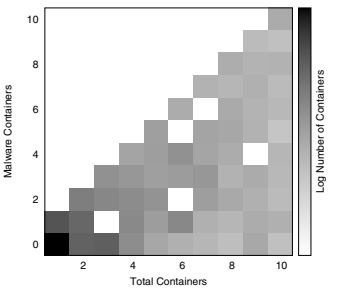



**Fig. 9.** Heatmap of Containers Associated with False Positives

### 5.3 Validity of Results under Biased Sampling

Can the conclusions drawn from Figures 6 and 7 reflect the real world behavior of our classifier where the proportion of malware files will be different than the one we used for training? As we explain, this is true for logistic regression. First, notice that among each of the two classes, malware and benign, sampling was random. So we can say that by selecting many malware files we have just uniformly increased the probability of malware by a factor $c_1 > 1$ and have uniformly decreased the probability of benign by a factor $c_0 < 1$:

$$p(y_i = 1|x_i, \text{selection}) = c_1 p(y_i = 1|x_i)$$
$$p(y_i = 0|x_i, \text{selection}) = c_0 p(y_i = 0|x_i).$$

Our logistic classifiers model the log odds so we get

$$\log \frac{p(y_i = 1|x_i, \text{selection})}{p(y_i = 0|x_i, \text{selection})} = \log \frac{c_1}{c_0} + \log \frac{p(y_i = 1|x_i)}{p(y_i = 0|x_i)}.$$

In other words, the log odds we compute are indeed inflated by $\log \frac{c_1}{c_0}$. However *the effect of* $\log \frac{c_1}{c_0}$ *is constant across all files.* On the other hand the DET curves of Figures 6 and 7 only depend on the *order of the files*, according to their probabilities. This order is not affected by adding to each file the constant $\log \frac{c_1}{c_0}$. Hence we would have obtained the same figures even if we had trained our logistic classifier with a sample that contained the correct proportion of malware.

## 6 Discussions

In this section we discuss the assumptions of our system, the constraints under which it needs to work, and investigate some issues related to the notion of suspicious but not necessarily malicious files and containers. Our work is based on two main assumptions. First the relationships we manage to extract from the files contain useful information that is not already captured by our current baseline classifier. The second assumption is that we can extract some of this information with our current representation. To avoid being detected by our approach, a piece of malicious software has to first score low or moderately when scanned with the individual classifier and then has to be associated only with containers that themselves are not suspicious. Assuming that the malicious file bypasses the individual classifier, the best strategies to avoid detection by the relationship classifier is to submit the file by itself or with another previously unseen file that is undoubtedly benign. Currently, our relationship classifier will not do anything to files it cannot directly relate to previously seen files. For such a file we could first find an approximate match (using, say, locality sensitive hashes [1]) and use its relationships. Though such an approach would further reduce opportunities for code reuse and "malware libraries" for malware authors, it is beyond the scope of this paper. Our focus is to demonstrate that even a simple approach has immediate benefits.

A very limiting constraint with respect to the solutions we could employ for our task is the requirement for our system to be highly scalable. We have therefore only considered linear models simply because we cannot afford to train (or even run) more complicated models on the millions of executables in our database. Among the linear models we only presented results for logistic regression, but other methods like SVMs should perform similarly.

Even though the results in Section 5 demonstrate large improvements, our model is not perfect. Table 4 and Figures 8 and 9 indicate our model is still susceptible to FPs which are particularly worrisome to analysts. However while statistical models are subject to false positives in general, we believe that the definition of a false positive is not correct in a portion of these cases. Typically, analysts assign a "benign" label to files which cannot infect a computer. We argue that even if a file cannot be infectious it can still be suspicious if, say, we have only seen it co-occurring (i.e. being part of the same container) with files that have been determined to be malware. To handle this case, we believe that the model can be further improved by adding a third label, "malware related", to any file which cannot infect a computer by itself, has been found in some relationship with malware (contained, dropped), and has never been observed in any files encountered during the installation of a legitimate software package.

Finally, the algorithm we proposed in Table 1 can be thought of as one iteration of a more complicated scheme. Though we already argued that iterating this algorithm is tricky, it could provide useful information that is currently not captured by our model. For example, two iterations of our algorithm would capture information about co-occurence patterns of files in the same archives and would start building archive reputations.

## 7 Related Work

Malware classification has been a rich area for research, and Idika *et al.* provide a recent survey of the literature [9]. Early efforts on malware classification such as those of Schultz *et al.* [20] and Kolter *et al.* [12] focused on static analysis of the executable files. Features based on n-grams of byte sequences have been used in [18,19]. In [21] the authors perform sequence analysis of system calls and Christodorescu *et al.* propose detecting malware based on the semantics of the unknown file [5]. Results show improved detection of obfuscated malware compared to commercial anti-virus products. Chouchane *et al.* [4] develop static classifiers for metamorphic malware, by computing probability distributions over metamorphic variants and using them to train a classifier. Perdisci *et al.* [17] proposed using boosting to classify malware.

Recently, a number of authors have proposed behavior-based malware detection systems. For example in [15] the system executes malware in a virtual machine allowing the execution of arbitrary programs at each control flow decision point. This allows the tool to explore, record, and report the complete behavior space of a program. Instead of using a simulated virtual machine for monitoring program behavior, Bayer *et al.* [2] developed a tool where a complete operating system is run in software thereby allowing the identification of malware that terminates after detecting that it is running in a virtual machine environment. Mehdi *et al.* [14] have previously used N-grams of system calls for a malware classification system.

Few papers have explored using graph-based methods for detecting malware. For example, [6,8,10] classify or cluster the call-graphs of malware and benign programs. Recently, Chau *et al.* [3] explored building file reputation based on a bipartite graph of applications and machines. Finally, Ye *et al.* [22] built and deployed a system that combines individual predictions with a different definition of file relationships. They consider two files related if they co-occur on a set of client machines. In contrast, we define our file relationships at the time a file is submitted to our service. Besides avoiding thorny privacy issues, the relationships we use are much more localized and reflect pieces of information that the human analysts actually seek when analyzing an unknown sample. In accordance to our experimental results, they also observe a large benefit by moving beyond individual file classification.

## 8 Conclusions

Automated malware detection is critical given the explosion of new malware in recent years. In this paper we investigate a novel way of improving malware classifiers by going beyond individually classifying a given file. Instead, we take advantage of the information that exists in the *relationships* between the files submitted to our service; information that is already being leveraged by human analysts in their job. Starting from a baseline individual file classifier we proposed three ways to propagate information from files to containers (Max neighbor,

Union bound and Biased logistic regression) and a relationship classifier which uses this propagated information to improve the baseline probabilities.

Our experiments show that on one hand simple approaches like the Union bound fail completely, and hence propagating and aggregating the relationship information is not a trivial task. On the other hand our proposed Biased logistic regression classifier completely dominates Max neighbor. Furthermore, using the container probabilities from Biased logistic regression the relationship classifier substantially reduces the FN rate at small FP rates.

In spite of these encouraging results, more algorithmic improvements are required to rely solely on automated malware classification to block or detect new malware. Improving the baseline classifier, which is relatively simple, can further improve the relationship classifier. In fact we have presented some evidence that large fractions of what appear to be false positives a) cannot be fixed by the relationship classifier and might be due to the baseline system and b) are not exactly false positives because they are files mostly found in malware containers. In any case, we believe that our modular approach of learning a correction to the baseline system nicely decouples the problem in two orthogonal components: one that looks at the file individually and one that looks at the file's relationships. This way, progress in either of these fronts can further improve the overall performance of our system.

**Acknowledgment.** The authors would like to gratefully thank Dennis Batchelder for helpful discussions and support.

## References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: 47th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2006, pp. 459–468. IEEE (2006)
2. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A tool for analyzing malware. In: Proceedings of 15th Annual Conference of the European Institute for Computer Antivirus Research, EICAR (2006)
3. Chau, D., Nachenberg, C., Wilhelm, J., Wright, A., Faloutsos, C.: Polonium: Tera-scale graph mining and inference for malware detection. In: Proceedings of SIAM International Conference on Data Mining, SDM 2011 (2011)
4. Chouchane, M., Walenstein, A., Lakhotia, A.: Statistical signatures for fast filtering of instruction-substituting metamorphic malware. In: Proceedings of the Workshop on Recurring Malcode, WORM 2007, pp. 31–37 (2007)
5. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, pp. 32–46 (2005)
6. Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X.: Synthesizing near-optimal malware specifications from suspicious behaviors. In: IEEE Symposium on Security and Privacy, pp. 45–60 (2010)
7. Golub, G.H., Loan, C.F.V.: Matrix Computations, 3rd edn. The Johns Hopkins University Press (1996)

8. Hu, X., Chiueh, T., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: ACM Conference on Computer and Communications Security, pp. 611–620 (2009)
9. Idika, N., Mathur, A.: A survey of malware detection techniques. Tech. rep., Purdue Univ. (February 2007), http://www.eecs.umich.edu/techreports/cse/2007/CSE-TR-530-07.pdf
10. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. Journal in Computer Virology, 33–45 (2011)
11. Kirda, E., Kruegel, C.: Behavior based spyware detection. In: Proceedings of the 15th USENIX Security Symposium, pp. 273–288 (2006)
12. Kolter, J., Maloof, M.: Learning to detect and classify malicious executables in the wild. Journal of Machine Learning Research, 2721–2744 (2006)
13. Manning, C.D., Raghavan, P., Schutze, H.: An Introduction to Information Retrieval. Cambridge University Press (2009)
14. Mehdi, S., Tanwani, A.K., Farooq, M.: Imad: in-execution malware analysis and detection. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 1553–1560 (2009)
15. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the IEEE Symposium on Security and Privacy, SP 2007, pp. 231–245 (2007)
16. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Proceedings of the 23rd Annual Computer Security Applications Conference, ACSAC 2007, pp. 421–430 (2007)
17. Perdisci, R., Lanzi, A., Lee, W.: Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In: Proceedings of the 2008 Annual Computer Security Applications Conference, ACSAC, pp. 301–310 (2008)
18. Krishna Sandeep Reddy, D., Dash, S.K., Pujari, A.K.: New Malicious Code Detection Using Variable Length $n$-grams. In: Bagchi, A., Atluri, V. (eds.) ICISS 2006. LNCS, vol. 4332, pp. 276–288. Springer, Heidelberg (2006)
19. Reddy, D., Pujari, A.: N-gram analysis for computer virus detection. Journal in Computer Virology, 231–239 (2006)
20. Schultz, M., Eskin, E., Zadok, E., Stolfo, S.: Data mining methods of detection of new malicious executables. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp. 38–49 (2001)
21. Sung, A., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: Proceedings of the 20th Annual Compter Security Applications Conference, ACSAC 2004, pp. 326–334 (2004)
22. Ye, Y., Li, T., Zhu, S., Zhuang, W., Tas, E., Gupta, U., Abdulhayoglu, M.: Combining file content and file relations for cloud based malware detection. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 222–230. ACM (2011)

# Understanding DMA Malware

Patrick Stewin and Iurii Bystrov

Security in Telecommunications — Technische Universität Berlin
Ernst-Reuter-Platz 7, 10587 Berlin, Germany
patrickx@sec.t-labs.tu-berlin.de,
l.inc@mailbox.tu-berlin.de
http://www.sec.t-labs.tu-berlin.de/

**Abstract.** Attackers constantly explore ways to camouflage illicit activities against computer platforms. Stealthy attacks are required in industrial espionage and also by criminals stealing banking credentials. Modern computers contain dedicated hardware such as network and graphics cards. Such devices implement independent execution environments but have direct memory access (DMA) to the host runtime memory. In this work we introduce DMA malware, i. e., malware executed on dedicated hardware to launch stealthy attacks against the host using DMA. DMA malware goes beyond the capability to control DMA hardware. We implemented DAGGER, a keylogger that attacks Linux and Windows platforms. Our evaluation confirms that DMA malware can efficiently attack kernel structures even if memory address randomization is in place. DMA malware is stealthy to a point where the host cannot detect its presense. We evaluate and discuss possible countermeasures and the (in)effectiveness of hardware extensions such as input/output memory management units.

**Keywords:** Dedicated Hardware, Direct Memory Access, I/OMMU, Keylogger, Malware, Manageability Engine, Rootkit, Stealth, vPro, x86.

## 1 Introduction

Recently the arms race between malware developers and the anti-malware community reached a new level. Countermeasures for kernel level [16], hypervisor based [20], and system management mode based malware [12] were proposed [13,26,5]. As a result researchers explored new environments for stealthy malicious software.

Malware can be placed on dedicated hardware such as video cards and network interface cards to attack the host platform [30,31,11]. Such devices bring, among other things, their own processor and runtime memory. These devices can operate independently from the host system. Anti-virus software cannot detect malicious code stored in separate memory and executed on a different processor.

An attacker can use such devices, or more precisely a mechanism called *Direct Memory Access* (DMA), to easily circumvent protection mechanisms built into the *Operating System* (OS) by attacking host runtime memory directly.

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 21–41, 2013.

We call code performing targeted DMA based stealthy attacks to find and read or modify target data *DMA malware*. Such data can be cryptographic keys for encrypted harddisks, credentials for online banking accounts, instant messenger chat sessions, and open documents located in the file cache.

In this paper we classify DMA attacks and derive the term DMA malware. We explore the term in more detail by examining if DMA malware can significantly increase the probability of performing a successful stealthy attack against a computer platform while preserving efficiency and effectiveness. For the evaluation we built our DMA malware DAGGER – a DmA based keystroke loGGER that exfiltrates captured data to an external entity. We are interested in the efficiency, effectiveness and especially in stealth properties of DMA malware. We chose to implement a keystroke logger to demonstrate that "short living" data can be captured by DMA malware.

Our implementation is based on *Intel's Manageability Engine* (ME) that is part of the popular x86 platform. Intel's ME is implemented in business as well as consumer platforms to support different applications, such as the *Intel Active Management Technology* (iAMT) [21] or the *Identity Protection Technology* (IPT) [19] (see Intel vPro platforms [18], for example).

Our DMA malware DAGGER is not executed on the host processor. It is executed on the processor provided by Intel's ME. No additional hardware is required. DAGGER implements a sophisticated isolated runtime attack on user input. Additionally, our DMA malware could steal cryptographic keys, target OS kernel structures in an attack, and copy files from the file cache.

Although DMA malware cannot by detected by anti-virus software, an attacker still faces certain challenges. DMA malware must be effective, i. e., it should be able to successfully attack various systems. DMA malware must also be efficient, i. e., fast enough to find and process data, even when dealing with virtual memory addresses and randomly placed data. Such malware goes beyond the capability to exploit DMA hardware.

The main contributions of this work are:

- **DMA Malware Definition.** There are different kinds of code that utilizes DMA. To clearly identify if code should be considered harmless, an attack, or DMA malware, we introduce an appropriate definition.
- **DMA Malware Core Functionality.** We present a number of requirements that must be fulfilled by DMA malware in order to mount successful attacks.
- **Evaluation of DMA Malware Prototype Implementations.** To prove that DMA malware increases the probability for successful stealthy attacks while preserving efficiency and effectiveness, we implemented DAGGER. DAGGER is executed on Intel's isolated ME. DAGGER operates stealthily and can attack multiple operating systems. Our implementation is so fast and efficient that it can capture keystrokes very early in the platform boot process, that enables DAGGER to capture harddisk encryption passwords under Linux, for example.

*Paper Organization.* Section 2 introduces necessary background. Our assumptions and attacker model are presented in Section 3. A classification of DMA code and a definition for DMA malware is given in Section 4. In Section 5 we present DMA malware core functionality. The design and implementation of our DMA malware is presented in Section 6. Section 7 describes the evaluation of DAGGER, Section 8 considers countermeasures and discusses in particular I/OMMU issues, and Section 9 presents related work. We conclude in Section 10.

## 2 Technical Background and Preliminaries

The target platform for our evaluation is a modern Intel x86 based system. This section introduces the most important terms regarding the target platform.

### 2.1 Typical x86 System Architecture

The main components of a typical x86 system architecture as depicted in Figure 1 are a *Central Processing Unit* (CPU or host processor), a *Memory Controller Hub* (MCH, also known as northbridge) and an *Input/output Controller Hub* (ICH, also known as southbridge). The combination of CPU, MCH, ICH is called the chipset [14]. System memory (*Random Access Memory* or in short RAM) as well as a display adapter are connected to the MCH. The MCH controls access to memory. It can block requests to memory addresses or redirect the request to the ICH, if the destination address belongs to the ICH. Peripheral devices, such as flash memory, *Network Interface Card* (NIC), etc., are integrated into the system using the *Peripheral Component Interconnect express* (PCIe) standard. This standard implements a serial interconnect for peripherals and the chipset. NICs and other add-on cards can be connected to the ICH via PCIe. Further controller devices connect other formats, such as *Universal Serial Bus* (USB), *FireWire*, or *Serial Advanced Technology Attachment* (SATA), via PCIe to the system. Legacy PCI devices are connected to the PCIe architecture via a so called *PCI-to-PCIe bridge* [4]. In Laptop computers *Personal Computer Memory Card International Association* (PCMCIA)/*ExpressCard* devices are integrated into the system utilizing PCIe.

The host CPU is not necessarily the only processor in the system. The video card, for example, supports a *Graphics Processing Unit* (GPU) to efficiently modify computer graphics. Data to be processed is stored in *Video RAM* (VRAM), that is separated from normal system RAM. Other devices with similar properties are NICs and Intel's Manageability Engine in the platform's MCH. They also utilize separate processors as well as separate RAM to execute firmware.

### 2.2 Direct Memory Access

PCIe supports DMA for peripherals for fast memory access without the involvement of the host CPU. The aim of DMA is to remove the burden from the

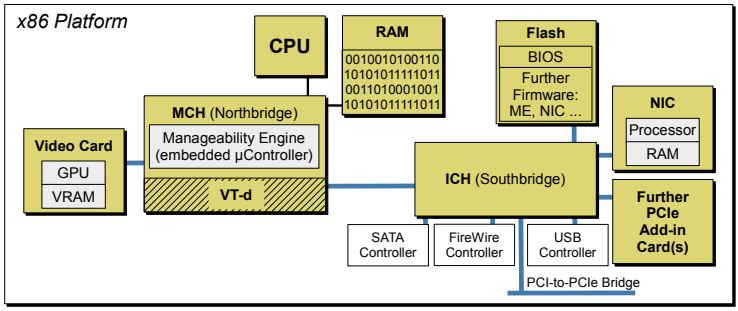



**Fig. 1.** x86 Chipset and Peripheral Components

host CPU. DMA allows peripherals to gain access to the whole host memory bypassing the CPU. The CPU can perform other tasks while DMA transfers occur. Peripherals can have their own engines to perform DMA. This kind of DMA is called first-party DMA [29, p.428]. Another mechanism is third-party DMA [29, p.428] where a central *DMA Controller* (DMAC) is necessary to provide legacy devices without DMA engines with fast memory access. It is also integrated in modern platforms [17, p.128].

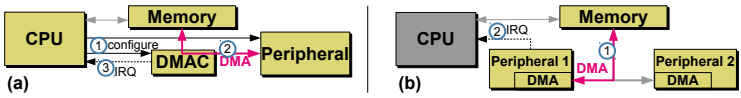



**Fig. 2. (a)** Third-party DMA: The host CPU (1) configures (source and destination address) the central DMA controller to (2) perform a DMA tranfser. The DMA controller (3) interrupts the host CPU when the DMA transfer has been finished [15, p.700]. Hence, the host CPU is aware of a third-party DMA transfer. — **(b)** First-party DMA: The peripheral device can (1) configure its own DMA engine. The device acts as bus master to get control of the system bus to perform a DMA transfer. The device can interrupt the host CPU when the device (2) has completed the transfer. The transfer also works if the device does not interrupt the host CPU at the end of the DMA transfer. In this case the CPU is completely unaware of the DMA transfer.

Figure 2 highlights an important difference regarding stealthy operation between third- and first-party DMA. When using third-party DMA the host CPU is aware of the DMA transfer, when using first-party DMA the host CPU is not necessarily aware of the transfer.

Note, a DMAC or a DMA engine can only access host memory addresses, but not host CPU cache, host CPU registers, or the harddisk, for example. The latter implies that data swapped out from runtime memory to the harddisk is not accessible by a DMA engine, either.

### 2.3 Input/Output Memory Management Units

Intel introduced a technology called *Intel Virtualization Technology for Directed I/O* (VT-d) [1] as one of several building blocks to provide hardware supported virtualization for x86 systems. VT-d can be considered as an *Input/Output Memory Management Unit* (I/OMMU) to efficiently assist virtualization requirements, such as reliable isolation of virtual machines running on a virtual machine monitor. VT-d is mainly used in conjunction with virtualization solutions. With VT-d, system software, that means a hypervisor or an OS, can create memory protection domains. For example, isolated subsets of physical memory can be assigned to a virtual machine or to memory of an I/O device driver. An I/O device not assigned to a protection domain has no access to physical memory of that domain. These access restrictions are realized using address translation tables. System software configures so called *DMA Remapping* (DMAR) engines provided by Intel VT-d. Such an engine maps a memory request, for example triggered by an I/O device, to physical memory. VT-d can block a memory request, if the device is not assigned to the protection domain.

## 3 Assumptions and Attacker Model

The attacker model describes the setting of a stealthy DMA attack scenario. The attacker is able to infiltrate dedicated hardware present in a computer platform with malicious payload remotely. This can be carried out via an OS or firmware related zero-day exploit [11], for example. The dedicated hardware supports DMA as described in Section 2. We assume that this computer platform has usual up to date defense mechanisms such as anti-virus software and a host firewall. The platform user does not apply additional hardware such as a hardware firewall to protect the computer platform.

We assume that only a completely stealthy attack can result in a successful attack. Hence, the attacker wants to hide the attack by using the stealth potential of dedicated hardware. Additional hardware would decrease the probability of a successful stealthy attack significantly. Most likely, the attacker aims on stealing data, e. g., to conduct industrial espionage or to aquire online banking credentials, etc.

## 4 DMA Malware Definition

To determine a definition for the term DMA malware we first classify different kinds of DMA based code. This helps to clearly distinguish between simple DMA usage, DMA attacks and DMA malware, whereby the latter has a clear focus on stealthiness. Note, DMA malware goes beyond the capability of controlling a DMA engine.

DMA based code implementing malicious functionality is considered as serious threat. Such code can be operating stealthily during infiltration and runtime.

It is also an advantage, e. g., for long-term attacks, if the code can survive platform reboots and power off as well as standby modes. Hence, we can prioritize the following criteria for our classification system. That is, the DMA based code:

- (C1) implements malware functionality
- (C2) needs no physical access to increase the probability of stealthy infiltration
- (C3) applies rootkit/stealth capabilities during runtime
- (C4) can survive reboot/standby/power off modes

With this prioritization we can derive a binary based classification:

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \mathsf{C1} & \mathsf{C2} & \mathsf{C3} & \mathsf{C4} \end{array}$$

This classification system covers 16 classes of DMA based code. We can derive a unique number for each class. For example, DMA based code that does not perform malicious actions ($\mathsf{C1} = 0$), leaves no traces on the host ($\mathsf{C3} = 1$), does not need physical access ($\mathsf{C2} = 1$), and cannot survive reboots ($\mathsf{C4} = 0$) is classified with the binary pattern 0110, that is class 6 in decimal. The higher the class, the more dangerous is the DMA based code. Note, we use this classification system to compare related work in Section 9.

Our definition of DMA malware is as follows:

> **Definition:** DMA malware is malicious software executed on dedicated hardware attacking a computer system via a mechanism called direct memory access as well as fulfilling at least the criteria C1, C2, and C3.

When applied to the target platform introduced in Section 2, this definition means, that DMA malware is based on first-party DMA and the DMA engine can be configured by the attack code to not involve the host CPU. The attack code is executed on dedicated hardware with its own processor and runtime memory, such as a NIC. Controlling the NIC increases the probability that an attacker can hide data during exfiltration.

## 5 DMA Malware Core Functionality

When attacking the host, it is not enough for an attacker to control a DMA engine. The engine enables the attacker to read and to write to host memory. However, in most cases the target memory address is not known.

**Overcoming Address Randomization.** The attacker has to determine memory addresses. The problem is that the memory space allocated for, e. g., kernel data structures is not at the same memory address after a platform reboot. Data structures are placed *randomly in memory* by the OS. This can happen

in a natural way when a device driver, for example, allocates memory and gets the next free unallocated memory chunk. The memory address of that chunk is not necessarily the same after a platform reboot. Alternatively, the OS can apply certain randomization algorithms to ensure that data structures are not placed at the same memory position. Of course, an attacker can scan the whole system memory for signatures of the target data, but this is very inefficient when scanning a system with 4 GB physical memory or more.

**Memory Mapping.** Operating systems work with *virtual memory addresses* [6, Chapter 15], but DMA works with *physical memory addresses*. The OS creates so called page tables that are used by the host CPU to map virtual memory addresses to physical ones. The mapping is absolutely necessary to resolve memory address pointers when using DMA. A special host processor control register called `CR3` contains the physical memory address of the page tables. The attacker has no access to the `CR3` register. The visibility of a DMA engine is restricted to host memory only.

**Search Space Restriction.** Without further investigations the attacker has to scan the whole memory address space for valuable data. There are two potential ways in which an attacker can overcome this problem. The first way is to analyze if the OS places the data structures in question in approximately the same memory area. The second possibility is to implement OS memory management mechanisms. That is, the attacker must find a way to access memory page tables created by the OS. With access to the page tables the attacker can then traverse page tables and is able to resolve pointers from one data structure to another. Note, this still requires a known starting point for the search.

## 6 Design and Implementation of DAGGER

We present an overview of a general design for our DmA based keystroke loGGER DAGGER in the next subsection before we explain the details of the DAGGER implementation in Subsection 6.2.

### 6.1 General Design

Our design of DAGGER is depicted in Figure 3. DAGGER is DMA malware. That is, DAGGER has to fulfill the DMA malware definition including at least the criteria `C1`, `C2`, and `C3`.

DAGGER consists of three main components. **Search:** find the address of valuable data in the host memory via DMA. **Process Data:** read valuable data within the regions identified during the search process. **Exfiltration:** exfiltrate information in a way that is invisible to the host.

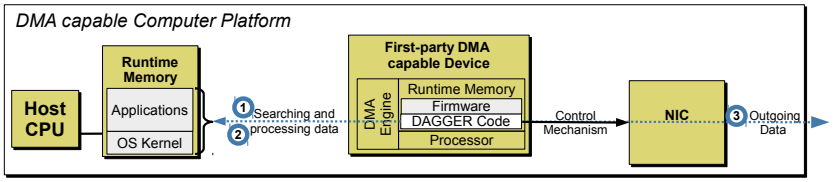



**Fig. 3.** General Design: DAGGER is executed on a DMA capable device so that it can (1) search and (2) process data from host runtime memory. It (3) controls a communication path to exfiltrate information.

### 6.2 Implementation Based on Intel's ME Environment

To evaluate DMA malware we chose to implement DAGGER on Intel's ME. Intel's ME provides some useful features for implementing DMA malware that we describe in the following paragraphs.

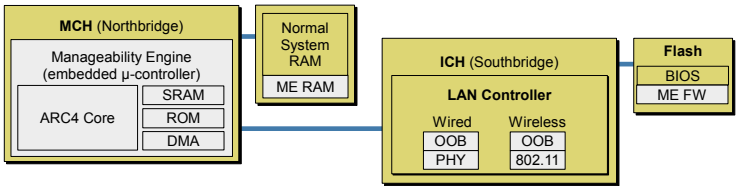



**Fig. 4.** Intel's Manageability Engine Environment

*Embedded µ-Controller.* The core of Intel's ME is an embedded µ-controller placed in the platform's MCH. This isolated environment contains *Read Only Memory* (ROM), *Static Random Access Memory* (SRAM), DMA hardware to access the host memory [5,28], and a processor as depicted in Figure 4. The embedded processor of the ME is an ARCtangent-A4 (ARC4). The isolated execution environment is available regardless of the power state, even in standby or power on/off. It only requires that the chipset is connected with a power source.

*ME Firmware.* Applications executed on the embedded µ-controller are implemented in firmware (ME FW) and stored in flash memory together with the BIOS. The most prominent ME firmware example is Intel's Active Management Technology [21]. But depending on the kind of computer platform (business or consumer hardware) the ME can also run other firmware. Other firmware executed by Intel's ME are for instance: Intel's Identity Protection Technology [19], *Alert Standard Format* [28, p.46], *Intel Quiet System Technology* for temperature and fan control [28, p.46], and *Integrated Trusted Platform Module* [21, p.109]. ME firmware can communicate with the host via a PCI device interface called

*ME Interface* (MEI) [21, p.71]. The MEI can provide the version of the executed ME firmware, for example.

*Separate Memory.* During the initial platform power-on procedure the ME firmware image is loaded into ME RAM. The firmware itself runs on the $\mu$-controller internal ARC4 processor and it also uses some system RAM as depicted in Figure 4 to store runtime data. This runtime storage is provided by a certain memory area that is invisible to the main CPU and the OS. The separation is enforced by the chipset [21].

*Out-of-Band Network Channel.* The ME environment introduces *Out-Of-Band* (OOB) communication, i. e., a special network traffic channel used by iAMT. The iAMT enabled computer platform is managed by a remote management console using OOB. OOB is also available regardless of the power state. OOB can be considered to be a separate network connection, running on the same hardware. The ICH implements necessary components to support the ME environment with the OOB feature. The firmware filters network traffic intended for, e. g., iAMT and redirects the packets to the ME. This kind of traffic is identified by TCP port numbers.

### 6.3 Attack Implementation Details for Linux and Windows Targets

We implemented two keystroke logger prototypes to attack two targets, Linux and Windows based OSes. We decided to find and monitor the keyboard buffer address of 32 bit versions of the target OSes. In comparison to 64 bit versions, 32 bit versions have to deal with a more complicated memory management. For example, the attacker has to consider *Physical Address Extensions* (PAE) [25, p.769] or certain memory offsets when mapping memory addresses. The following subsections describe, how we implemented the DMA malware core functionality as described in Section 5. The prototypes capture short living keystroke codes within their *monitoring phase*. Each prototype handles the *search phase* for the target buffer differently. This has at least two reasons. One reason is to evaluate as many aspects as possible of DMA malware. The other reason is that OSes have different memory management properties.

We use a vulnerability described in [28] to infiltrate the ME environment during runtime. To call our code we hook a ME firmware function that we identified as the library function `memset`. The authors of [28] assumed to hook a timer interrupt handler. But actually they hooked the ME firmware function `memcpy`. We hook `memset` since we determined that it is called more often.

**Linux.** Our Linux variant is based on a signature scan as depicted in Figure 5. We analyzed the available Linux source code to derive a signature of our target, the physical address of the keyboard buffer. The address of the buffer is part of the *USB Request Block* (URB) structure that is defined in the file `include/linux/usb.h` of the Linux source code. The demanded structure field is called `transfer_dma`. The memory offsets differ from kernel version to kernel version. We solved that problem by exploiting the *Grand Unified Bootloader*

(GRUB) that places a kernel identifer at a constant physical memory address. We implemented a function that reads the identifier via DMA and parses the kernel version number to derive corresponding offsets. Afterwards our prototype runs through the search phase, that is, the signature scan.

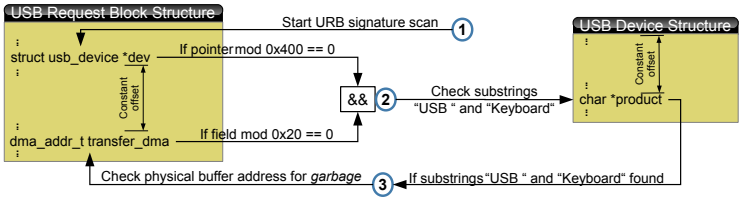



**Fig. 5.** USB Request Block Signature Scan (simplified): The scan (1) begins to search for a pointer to the USB device structure. A candidate for such a pointer is aligned to a `0x400` boundary. The structure field `transfer_dma` must be aligned to a `0x20` boundary. If both conditions are true, the product string in the USB device structure is (2) checked for the substrings "USB " and "Keyboard". In the last step the signature scan (3) checks if the keyboard buffer contains *garbage*, that is, invalid keystroke codes.

Since our Linux prototype targets kernel data structures we can restrict the search space to the first gigabyte of system RAM. Standard Linux systems have a memory split of 1 GB/3 GB, that means, 1 GB for kernel space and 3 GB for user space. We were able to further restrict the search space by empirically analyzing in which memory area the kernel places the data structures needed by our signature scan. We determinded that this memory area is between `0x33000000` and `0x36000000` for the Ubuntu Linux kernel version 3.0.0 after a fresh platform boot. The address of the keyboard buffer does not change after standby or hibernate mode. With this approach we overcome the problem of inefficiently scanning the whole system memory for the randomly placed signature. Mapping virtual adddresses to physical ones is a minor issue when attacking the Linux kernel. Normally, in 32 bit versions a kernel virtual address (or more precisely kernel logical address [6, Chapter 15]) is mapped to its physical address by substracting a constant offset. In 64 bit Linux versions such an offset is not needed. Hence, there is no need to know the content of the `CR3` processor register.

**Windows.** To be able to perform the search using the search path as described below, virtual addresses must be mapped to physical ones. This mapping is done using page tables created by the Windows kernel. The memory address of those page tables is loaded into the `CR3` register, which an attacker cannot access via DMA. It turned out after some empirical tests with a simple driver, that the physical address of the page tables for the *system process* takes one of the following two values for Windows Vista/7 systems: `0x122000` or `0x185000`. The system process is the first process created during Windows startup. With this knowledge DAGGER can access the page tables created by the kernel and

overcomes the problem of mapping virtual addresses to physical ones. DAGGER implements a page table traversing algorithm that takes account of PAE.

Our Windows sample searches for a structure called `DeviceExtension` that is maintained by the USB keyboard driver `kbdhid.sys`. This structure contains a buffer that stores the codes of the last pressed keys. The source code for `kbdhid.sys` is not publicly available. The most convenient way to get internal information of that driver was to use *IDA Pro*[1], *Windows Debugger* (WinDbg) tools, and debug symbols provided by Microsoft[2] in form of `pdb` files.

To finally determine the location of the buffer in the `DeviceExtension` structure, our research starts quite early in the Windows boot process [25, Chapter 13]. We analyzed further internal Windows structures. To find a starting point for the search, we analyzed the *Kernel Processor Control Region* (KPCR [25, p.62ff]), or more precisely `KiInitialPCR`, the KPCR for the processor 0. We also examined the *Object Manager Namespace Directory* (OMND, part of the Windows object manager). We figured out that `KiInitialPCR` is well suited to derive a path to the `DeviceExtension` structure as depicted in Figure 6.

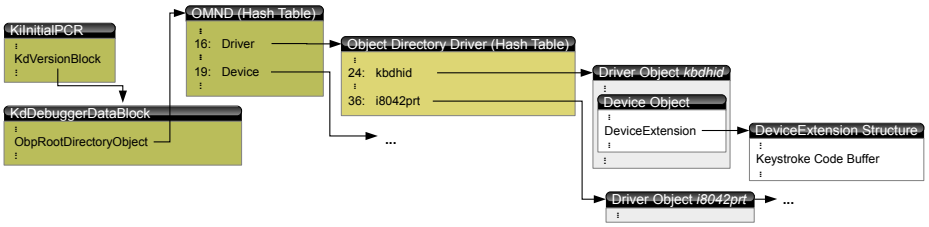



**Fig. 6.** Find `DeviceExtension` Structure (simplified): With `KiInitialPCR` as a starting point, DAGGER finds the OMND, that provides via hash tables a path to the driver object `kbdhid`. This object contains a pointer to a device object. The device object provides the `DeviceExtension` structure, which contains the keystroke code buffer.

`KiInitialPCR` is not located at a constant memory address. DAGGER has to apply another step before it can start with the search as depicted in Figure 6.

The memory position of `KiInitialPCR` is determined by a function called `OslpLoadAllModules` of the `winload.exe` binary as depicted in Figure 7. This binary is loaded by the Windows boot manger `bootmgr` that in turn is loaded by *Master Boot Record* (MBR) code, etc. The function loads the *Hardware Abstraction Layer* (HAL) library `hal.dll` as well as the Windows kernel image in a more or less random manner. The kernel image contains `KiInitialPCR` at a constant relative address. The disassembled code of `OslpLoadAllModules` is reminiscent of an *Address Space Layout Randomization* (ASLR) mechanism [25, p.757].

[1] See `http://www.hex-rays.com/products/ida/index.shtml`
[2] See `http://msdn.microsoft.com/en-us/windows/hardware/gg462988`

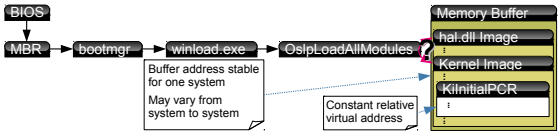



**Fig. 7.** Find `KiInitialPCR` (simplified): `OslpLoadAllModules` determines the exact position of the Windows kernel image and the HAL

The memory buffer for the kernel image and the HAL is allocated by `OslpLoadAllModules` via a function called `BlImgAllocateImageBuffer`. The latter function returns stable address values for a Windows system. These values may vary on different systems. For every possible return value of the function `BlImgAllocateImageBuffer` there are 64 theoretically possible different 4 KB aligned virtual addresses. These addresses need to be checked in order to find the kernel image base address. The disassembly of `OslpLoadAllModules` revealed that the randomization seed for the address randomization has a 5 bit value. This implies 32 possible addresses for each (of two) possible load order cases, i. e., first kernel image and then `hal.dll` or vice versa. As long as `KiInitialPCR` has a constant relative virtual address within the kernel image, the same number of virtual addresses to be checked also applies for a direct `KiInitialPCR` search without any need to deal with the kernel image. To ensure that DAGGER found the correct `KiInitialPCR` we implemented a `KiInitialPCR` signature check. When DAGGER has found the correct `KiInitialPCR`, DAGGER continues to look for the keyboard buffer using the search path described in Figure 6.

## 7 Evaluation

We used an x86 platform with a Q35 chipset, 2 GB RAM, a 4-core 3 GHz CPU, and iAMT firmware (version 3.2.1) to evaluate DAGGER with four different 32 bit OS kernels: Windows Vista Business (Service Pack 2), Windows 7 Professional (Service Pack 1) and Ubuntu Linux kernel version 2.6.32 as well as kernel version 3.0.0. The DAGGER attack binary code has a size of approximately 33 KB for Linux and 31 KB for Windows.

**DMA Malware Fulfillment.** We designed and implemented our DAGGER prototypes according to the DMA malware definition described in Section 4. (C1) is clearly fulfilled since it implements working keystroke logger functionality. DAGGER needs no physical access for the infiltration process (C2). We infiltrate the ME environment using a software based exploit during runtime. DAGGER exploits dedicated hardware to implement rootkit properties (C3). We ran host performance overhead tests (memory: MEM, network: NET, and CPU), since host and ME environment share the NIC as well as a RAM chip. Parallel NIC and RAM accesses must be arbitrated and could therefore cause delays.

Our measurement results depicted in Figure 8 reveal no significant overhead. The highest overhead that we could detect is approximately 1.5 % when accessing the host memory during the search phase. It is extremely unlikely that this minimal overhead would reveal DAGGER. The search times summarized in Figure 9 are very short and the very aggressive memory stress test we performed does not represent the memory utilization of a normal computer system.

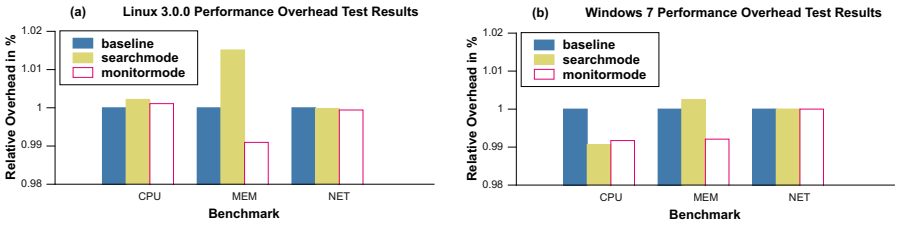



**Fig. 8.** Host Performance CPU, MEM, and NET Overhead Tests: We used *Time Stamp Counters* [6, p.186] to measure overhead time. We measured the time it takes to copy a 100 MB test file over the network (NET) and within RAM (MEM) as well as the time it needs to compute a SHA1 hashsum over this test file ten times in parallel to stress all four CPU cores (CPU). Each benchmark was performed three times: without keystroke logger (baseline), keystroke logger in search mode, and keystroke logger in monitoring mode. For the monitoring mode we configured the keystroke logger to constantly send network packets of approximately 1000 packets per minute. This is equal to 500 keystroke and 500 key release events. We repeated each test 1000 times. A bar in the figure represents the mean of 1000 runs.

DAGGER has solely read-only operations to ensure stealthiness. The popular network sniffer *Wireshark*[3] was not able to detect any DAGGER traffic on Linux and Windows systems. Host firewalls cannot block such traffic either. Even if anti-virus software knew DAGGER's signature it would be unable to access DAGGER's memory to apply the signature scan successfully. Nethertheless, we also run a software called *Mamutu*[4], that is, amongst other things, specialized in detecting keylogger behavior. Even specialized software could not find any indication of DAGGER. Regarding criterion C4 we successfully checked if DAGGER's attack code is fully functional after a platform reboot, after standby and after power off state. We determined that this depends on an iAMT BIOS option. Our code cannot survive a cold boot that happens if this option is not set.

**Effectiveness and Efficiency.** DAGGER is efficient, since it can permanently catch short living data from the keyboard buffer. To prove that DAGGER is also effective we tested DAGGER with different Windows and Linux versions as well as several keyboards (Logitech, Dell, FujitsuSiemens). The measured search

[3] See `http://www.wireshark.org/`
[4] See `http://www.emsisoft.com/en/software/mamutu/`

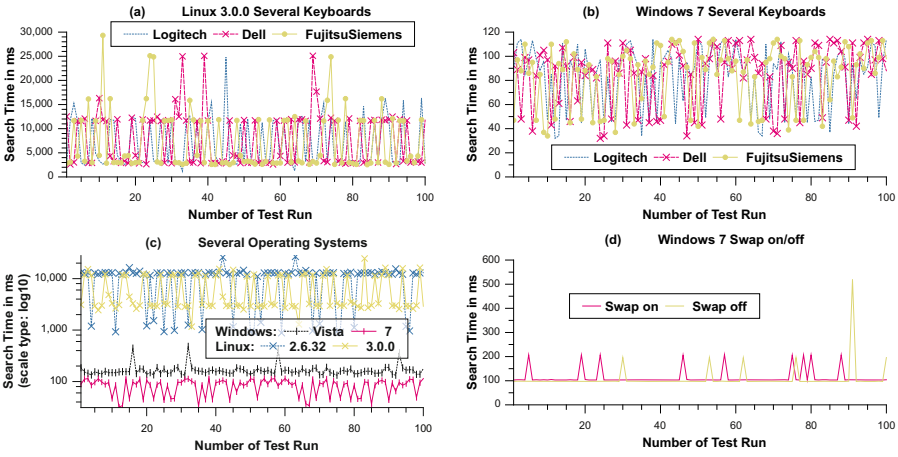



**Fig. 9.** Search Time Measurement Results: The test results with several keyboards under Linux reveal a best case for search times of around 1000 ms and a worst case of almost 30,000 ms as depicted in (a). The median for all keyboards is at 3281 ms. Useful for comparison: scanning the whole memory area determined for Linux (see Section 6.2) search takes approximately 13,000 ms. The worst case of 30,000 ms is due to an erroneous DMA transfer that we do not handle directly. This causes DAGGER to repeat the search phase. On Windows 7 the best search time is approximately 50 ms and the worst time is around 120 ms, see (b). The median for all keyboard is at 93 ms. Hence, the search strategy we implemented for Windows targets performs much better than the signature scan based strategy for Linux. The plot in (c) compares different target kernels. DAGGER performs slightly better for Windows 7 than for Windows Vista. Linux 2.6.32 places the target memory structure closer to `0x33000000` than Linux 3.0.0. Thus, DAGGER has more hits around 1000 ms when attacking Linux 2.6.32. The results in (d) confirm that swapping has no effect on the efficiency and effectiveness of DAGGER. A platform reboot was only applied to change the swapping behavior. The peaks are due to search phase repeats.

times summarized in Figure 9 prove that DAGGER is quite efficient. We repeated the measurements for each kernel and for each keyboard 100 times. We took a measurement after a platform (re)boot to change the target address for each test run. The Linux measurement results imply that we could further restrict the search space. We could start the search near the lowest address we encountered most often during our tests. Search times of around 2500 ms are due to target addresses near `0x33c00000`. Thus, we could skip almost 2500 ms if we start the search at `0x33c00000`. Furthermore, we could skip the search area address range between `0x34000000` and `0x36000000`. Almost no targets were found in this area. A lot of targets were found near `0x36e0000`, i. e., search times of around 12,500 ms that could also be saved. This increases the probability to miss keyboard buffer addresses. That is, we can get better (similar to the Windows attack) search times at the expense of effectiveness. The best case

search times are sufficient to capture hard disk encryption passwords, for example. We tested this successfully with a Linux system. The Windows kernel can swap out memory pages to the hard disk – Linux does not. Swapped memory pages cannot be found by DMA malware. Hence, we also did a test for Windows to check if swapping has any effect on DAGGER as depicted in Figure 9 (d).

**ME Firmware Condition.** To be really stealthy DAGGER ensures that the ME firmware is still up and running correctly. iAMT provides a webserver for remote platform management [21, p.215] that is still usable. The server responds correctly on the local platform on Linux and Windows. Firmware tools utilzing the MEI (see Section 6.2) also work when DAGGER is active. We successfully tested the *AMT Status Tool* (part of the *Local Manageability Service* driver) and the *Manageability Connector Tool* (part of the *Manageability Developer Toolkit 7.0*) under Windows. Under Linux we successfully tested the *Intel AMT Opensource Tools and Drivers* (version 5.0.0.30), or more precisely the *ME Status* and the *ZTCLocalAgent* tool. Note, we determined that DAGGER still runs when we deactivated the iAMT firmware in the BIOS. It appears that the ME environment cannot be disabled entirely via any BIOS options.

**I/OMMU.** To test an I/OMMU as a countermeasure against DAGGER we enabled Intel VT-d in the BIOS. As far as we know Windows does not support I/OMMUs directly. We could successfully attack Windows Vista and Windows 7 although the I/OMMU was activated. Linux experimentally supports I/OMMU configuration with additonal effort. We also enabled VT-d in the BIOS and we activated I/OMMU support via the kernel command line. With these additional steps we were able to prevent the Linux version of DAGGER from reading short living keystroke codes from OS memory. This protection is not activated by default and the code is still experimental. In the next section we discuss, among other things, further issues regarding the I/OMMU.

## 8 Countermeasures

To scan for DMA malware using software executed on the host CPU is quite difficult. For example, current AV software does not scan the runtime memory of peripherals or the host CPU cannot access the runtime memory due to certain isolation mechanisms. The worst case for a scanning approach is that the DMA malware changed the behavior of the scan software, which would deliver incorrect results. Checking firmware images at load time, as proposed by the *Trusted Computing Group* [32], does not prevent runtime attacks. Furthermore, it is unclear if all ROM components are accessible by the host.

**I/OMMU Issues.** In the case of DMA attacks an appropriate configuration of the I/OMMU (see Section 2.3) is proposed as a preventive countermeasure, for example in [11, p.48]. It is required that system software configures the I/OMMU. An incorrect configuration cannot be excluded [22, p.2].

It is assumed that the I/OMMU is secure. Unfortunately this is not always the case. The authors of [27] demonstrated that an I/OMMU configuration can be tricked with legacy PCI devices. In [35] is revealed that an I/OMMU can be attacked by modifying the number of DMA remapping engines provided by the BIOS (see Section 2.3). This is done before the I/OMMU is configured by system software. The environment we used for DAGGER is able to carry out such an attack. This threat can only be mitigated by executing special hardware dependent code called `SINIT`. However, on at least one previous occasion the manufacturer of the chipset failed to release `SINIT` code at the launch of the chipset [34, p.22]. This code is needed to initialize a well known and trustworthy environment for, e. g., a hypervisor. It checks the DMA remapping engines and can therefore prevent an attack as presented in [35]. `SINIT` belongs to and increases the size of the trusted computing base. Previous work demonstrated that `SINIT` code can have exploitable security vulnerabilities that can be used to trick I/OMMU mechanisms [35]. Recently, the authors of [33] presented another attack that can be used to circumvent I/OMMU mechanisms, too. To prevent the attacks presented in [35,33], a `SINIT` as well as a BIOS update must be applied. Another I/OMMU attack was presented in [34]. Note, `SINIT` is normally triggered on hypervisor based platforms. Platforms running a normal OS cannot necessarily count on the I/OMMU. It should also be mentioned that `SINIT` requires to activate additional platform features, namely the *Trusted eXecution Technology* and the *Trusted Platform Module* [14]. That means, users that do not want to activate the TPM for example cannot count on the I/OMMU either. Note, the TPM is an opt-in device [14, p.212] and is turned off by default.

For a comprehensive protection against DMA malware it is absolutely necessary to correctly configure the I/OMMU. However, the I/OMMU can only be considered secure if the above mechanisms to protect the whole platform are secure. This is a difficult task. Hence, alternative approaches were considered in [22] and [10]. The authors of [22] state that their approach requires extending the firmware, does not work correctly if peripherals cause heavy PCIe traffic, and the verifier component needs to know the exact hardware configuration. The approach presented in [10] is highly NIC adapter-specific and not applicable to isolated environments such as Intel's ME. It is worth noting that malware such as our implementation controls the NIC without any NIC firmware modifications, i. e., exfiltration cannot be detected by the approach described in [10]. Furthermore, this approach has significant performance issues for the host CPU (100 % utilization of one CPU core).

Memory access policies enforced by I/OMMUs can be insufficient or can even prevent the use of some other features in some application scenarios. Consider hardware supported malware scanners such as CoPilot [24] and DeepWatch [5]. The I/OMMU can be configured to stop CoPilot and DeepWatch from working or to allow such systems to access the host memory to scan it for malicious software. In the latter case DMA malware could make use of the execution environment of CoPilot or DeepWatch to attack the host. DAGGER, for example,

uses the DeepWatch environment, i. e., Intel's ME. Since iAMT version 5, Intel supports a verified launch for the firmware to be executed on Intel's ME [21, p.271]. The firmware is checked during load time. The result of the load time check is provided to system software. As far as we know the result is not used in practice. The mechanism cannot prevent runtime attacks as applied by our implementation. This means, DAGGER proves that our assumption that an attacker already infiltrated the target system, e. g., via a zero-day exploit (see Section 3), can also hold even if such additional security mechanisms are in place.

On the one hand an appropriate configuration of the I/OMMU is a first step against DMA malware. On the other hand, without resolving the mentioned issues a successful deployment cannot be guaranteed.

## 9 Related Work

The discussion of related work is based on the classification system and its criteria C1 – C4 that we introduced in Section 4.[5]

Since 2004 several DMA attacks using additional hardware such as USB devices [23], special PCMCIA cards [2], and Firewire devices [8,9,3] were presented. According to C2 those approaches cannot be considered as DMA malware. According to our classification system of Section 4 the attacks presented in [8,9,3,2] are classified in class 11 (1011). The attack presented in [23] is in class 1 (0001) since it reveals itself.

In [28] it was demonstrated that Intel's ME can be used to write to host memory. The authors of [28] described a vulnerability that allows to inject code into the ME environment. The code of [28] did not implement any malware behavior. It reveals itself by writing to a known hard coded host memory address. Hence, this approach is in class 5 (0101). Furthermore, it did not demonstrate how to read from host memory and how to use the OOB network channel.

On the contrary the attacks described in [30,31,11], and [7] fulfill all criteria for DMA malware. More precisely, they are classified in class 15 (1111). The authors of [30,31] presented a stealthy secure shell that offers memory inspection using DMA. A combination of NIC and video card is used to hide the shell. The shell is installed by reflashing firmware remotely. Our DAGGER prototype does not require to infilrate code into two peripherals and it does not require to reflash firmware. The attack presented in [11] exploits a vulnerability in the firmware of a NIC during runtime. The compromised NIC is used to attack the host system by adding a backdoor. The authors of [11] described how the host could access the NIC internal memory. This offers a possibility to detect the DMA malware using code executed on the host CPU. As far as we know no anti-virus like software makes use of this. It should be mentioned that the host access to the NIC internal memory is not a common feature. Normally, the runtime memory of the

[5] Note, all classifications were done using publicly available material. If we could not decide with the help of available resources whether a criterion is fulfilled, we assume that this criterion is fulfilled.

Intel ME environment used for our DAGGER implementation is not accessible by the host. The work of [7] is quite similar to [11]. Both attacks use the same NIC. The malware described in [7] aims to implement rootkit capabilities. Their work is still in progress.

## 10 Conclusion

In this work we studied DMA malware, i.e., malware hidden on dedicated hardware. Such malware can circumvent protection mechanisms run on the host CPU by directly accessing host memory. We implemented and evaluated DAGGER, a DmA based keystroke loGGER. The dedicated hardware enables our prototype to benefit from rootkit properties. DAGGER operates stealthily. It is undetectable by anti-virus software etc.

DMA malware is more than controlling a DMA engine. Our evaluation confirmed that DMA malware is quite efficient even if obstacles such as memory address randomization are in place. We also demonstrated that DMA malware can be quite effective, that is, it can attack several OSes. This verifies that DMA malware is stealthy at no costs regarding efficiency and effectiveness.

Currently, the host has no reliable means to protect itself. Throughout this work we highlighted that the I/OMMU has several issues and the host cannot necessarily count on this preventive countermeasure against DMA malware. Besides possible vulnerabilities and various preconditions that must be fulfilled for a successful I/OMMU deployment, the most obvious issue is that common OSes do not or do not sufficiently support the I/OMMU. Hence, currently, DMA malware can easily attack OSes such as Windows. A general and reliable approach for scanning the dedicated devices for malware does not exist. Future work is needed to develop a reliable and more general DMA malware detection mechanism. Until such a solution is developed, only dedicated hardware that is fully accessible by the host, i. e., complete RAM and ROM access, should be deployed. This enables the host to check the device for malicious modifications from time to time. A precondition for this is a reasonable measurement strategy and that the detector gets loaded first.

We conclude that dedicated hardware with a separate processor, runtime memory, and a DMA engine are a serious threat for the host platform. DMA malware executed on such devices is quite effective and efficient. DMA malware clearly demonstrates that additional protection mechanisms are needed to ensure a platform's confidentiality, integrity, and especially its trustworthiness.

**Acknowledgments.** The authors would like to thank Dmitry Nedospasov, Jean-Pierre Seifert and especially Collin Mulliner for useful discussions and their inevitable help to complete this paper. The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

## References


1. Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., Wiegert, J.: Intel Virtualization Technology for Directed I/O. Intel Technology Journal 10(3), 179–192 (2006)
2. Aumaitre, D., Devine, C.: Subverting Windows 7 x64 Kernel with DMA attacks. Sogeti ESEC Lab (July 2010), http://esec-lab.sogeti.com/dotclear/public/publications/10-hitbamsterdam-dmaattacks.pdf
3. Boileau, A.: Hit by a Bus: Physical Access Attacks with Firewire. Security-Assessment.com, Ruxcon 2006 (October 2006), http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf
4. Budruk, R., Shanley, T., Anderson, D.: PCI Express System Architecture. The PC System Architecture Series. Addison Wesley, Pearson Education, MindShare, Inc. (July 2010)
5. Bulygin, Y.: Chipset based Approach to detect Virtualization Malware. TuCancUnix (2008), http://www.tucancunix.net/ceh/bhusa/BHUSA08/speakers/Bulygin_Detection_of_Rootkits/bh-us-08-bulygin_Chip_Based_Approach_to_Detect_Rootkits.pdf
6. Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers, 3rd edn. O'Reilly Media, Inc. (2005)
7. Delugré, G.: Closer to metal: Reverse engineering the Broadcom NetExtreme's firmware. Sogeti ESEC Lab (October 2010), http://esec-lab.sogeti.com/dotclear/public/publications/10-hack.lu-nicreverse_slides.pdf
8. Dornseif, M.: 0wned by an iPod - hacking by Firewire. Laboratory for Dependable Distributed Systems University of Mannheim, PacSec 2004 (November 2004), http://pi1.informatik.uni-mannheim.de/filepool/presentations/0wned-by-an-ipod-hacking-by-firewire.pdf
9. Dornseif, M., Becher, M., Klein, C.N.: FireWire – all your memory are belong to us. CanSecWest (May 2005), http://cansecwest.com/core05/2005-firewire-cansecwest.pdf
10. Duflot, L., Perez, Y.-A., Morin, B.: What If You Can't Trust Your Network Card? In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 378–397. Springer, Heidelberg (2011)
11. Duflot, L., Perez, Y.-A., Valadon, G., Levillain, O.: Can you still trust your network card? French Network and Information Security Agency (FNISA) (March 2010), http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf
12. Embleton, S., Sparks, S., Zou, C.: Smm rootkits: a new breed of os independent malware. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Networks, pp. 1–12. ACM, New York (2008)
13. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. Network and Distributed Systems Security Symposium (February 2003)
14. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press (2009)
15. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann (May 2005)

16. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional (2005)
17. Intel Corporation: Intel I/O Controller Hub (ICH9) Family. Intel Corporation (August 2008), http://www.intel.com/content/dam/doc/datasheet/io-controller-hub-9-datasheet.pdf
18. Intel Corporation: 2nd Generation Intel Core vPro Processor Family. Intel Corporation (June 2011), http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf
19. Intel Corporation: Access Accounts More Securely with Intel Identity Protection Technology. Intel Corporation (February 2011), http://ipt.intel.com/Libraries/Documents/Intel_IdentityProtect_techbrief_v7.sflb.ashx
20. King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: SP 2006: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 314–327. IEEE Computer Society, Washington, DC (2006)
21. Kumar, A., Goel, P., Saint-Hilaire, Y.: Active Platform Management Demystified. Richard Bowles, Intel Press (2009)
22. Li, Y., McCune, J.M., Perrig, A.: VIPER: Verifying the integrity of peripherals' firmware. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS (October 2011)
23. Maynor, D.: DMA: Skeleton key of computing && selected soap box rants. CanSecWest (May 2005), http://cansecwest.com/core05/DMA.ppt
24. Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: Proceedings of the 13th Conference on USENIX Security Symposium, SSYM 2004, vol. 13. USENIX Association, Berkeley (2004)
25. Russinovich, M., Solomon, D.A.: Windows Internals: Including Windows Server 2008 and Windows Vista, 5th edn. Microsoft Press (2009)
26. Rutkowska, J.: Red Pill... or how to detect VMM using (almost) one CPU instruction. Internet Archive (November 2004), http://web.archive.org/web/20110726182809/, http://invisiblethings.org/papers/redpill.html
27. Sang, F., Lacombe, E., Nicomette, V., Deswarte, Y.: Exploiting an I/OMMU vulnerability. In: 2010 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 7–14 (October 2010)
28. Tereshkin, A., Wojtczuk, R.: Introducing Ring -3 Rootkits. Black hat (July 2009), http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf
29. Thompson, R.B., Thompson, B.F.: PC Hardware in a Nutshell, 3rd edn. O'Reilly & Associates, Inc., Sebastopol (2003)
30. Triulzi, A.: Project Maux Mk.II. The Alchemist Owl (2008), http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf
31. Triulzi, A.: The Jedi Packet Trick takes over the Deathstar. The Alchemist Owl (March 2010), http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf

32. Trusted Computing Group: TCG PC Client Specific Impementation Specification for Conventional BIOS. TCG (July 2005), http://www.trustedcomputinggroup.org/files/temp/64505409-1D09-3519-AD5C611FAD3F799B/PCClientImplementationforBIOS.pdf
33. Wojtczuk, R., Rutkowska, J.: Attacking Intel TXT via SINIT code execution hijacking. ITL (November 2011), http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf
34. Wojtczuk, R., Rutkowska, J.: Following the White Rabbit: Software attacks against Intel VT-d technology. ITL (April 2011), http://www.invisiblethingslab.com/resources/2011/Software20Attacks20on20Intel20VT-d.pdf
35. Wojtczuk, R., Rutkowska, J., Tereshkin, A.: Another Way to Circumvent Intel(R) Trusted Execution Technology. ITL (December 2009), http://invisiblethingslab.com/resources/misc09/Another20TXT20Attack.pdf

# Large-Scale Analysis of Malware Downloaders

Christian Rossow[1,2,⋆], Christian Dietrich[1,3], and Herbert Bos[2]

[1] University of Applied Sciences Gelsenkirchen, Institute for Internet Security, Germany
[2] VU University Amsterdam, The Network Institute, The Netherlands
[3] Department of Computer Science, Friedrich-Alexander University, Erlangen, Germany
{rossow,dietrich}@internet-sicherheit.de

**Abstract.** Downloaders are malicious programs with the goal to subversively download and install malware (*eggs*) on a victim's machine. In this paper, we analyze and characterize 23 Windows-based malware downloaders. We first show a high diversity in downloaders' communication architectures (e.g., P2P), carrier protocols and encryption schemes. Using dynamic malware analysis traces from over two years, we observe that 11 of these downloaders actively operated for at least one year, and identify 18 downloaders to be still active. We then describe how attackers choose resilient server infrastructures. For example, we reveal that 20% of the C&C servers remain operable on long term. Moreover, we observe steady migrations between different domains and TLD registrars, and notice attackers to deploy critical infrastructures redundantly across providers. After revealing the complexity of possible counter-measures against downloaders, we present two generic techniques enabling defenders to actively acquire malware samples. To do so, we leverage the publicly accessible downloader infrastructures by replaying download dialogs or observing a downloader's process activities from within the Windows kernel. With these two techniques, we successfully milk and analyze a diverse set of eggs from downloaders with both plain and encrypted communication channels.

**Keywords:** Malware, Downloader, Dropper, Dynamic Analysis.

## 1 Introduction

A crucial part in a malware's lifecycle is to spread, e.g., via spam, drive-by downloads or exploiting vulnerabilities. Whereas malware such as worms spreads on its own, attackers have begun to separate the task of infecting victim systems and the exploitation or "monetization" of the infected systems. Recent investigations to this business, known as "Pay-per-Install" (PPI), have shown the vast potential of this kind of malware distribution model. Caballero et al. [5] analyzed PPI networks by actively infiltrating and participating a handful of PPI programs. It was shown that PPI networks are responsible for installing a diverse set of malware on infected systems.

Technically, the PPI scheme is only a subset of the malware type that we term a *downloader*. A downloader is a malicious program with the purpose to subversively

⋆ We thank all malware sample providers and the VirusTotal team for their input. We thank Dennis Fricke and Andrea Lanzi for their support in developing the Windows kernel driver, and Arne Welzel for his reverse engineering efforts. This work was supported by the Federal Ministry of Education and Research of Germany (Grant 01BY1110, MoBE).

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 42–61, 2013.

download and install malware on a victim's machine. The specifics of PPI networks allow attackers to get paid on a per-system and per-affiliate basis, but the effect of PPI or, more generally, downloaders is comparable: Once a downloader is executed, and no matter if related to a PPI network or not, the running system will typically be compromised with multiple different malware families. Thus, downloaders represent a simple yet widely-used way to spread new malware, typically as part of a service model within the underground community.

In this paper, we outline and analyze the landscape of what we think represents a snapshot of prevalent and current downloaders. We identified 23 downloaders, of which many – to the best of our knowledge – have not yet been documented. We characterize these downloaders concerning their communication model. For example, we discuss the communication architectures of downloaders (e.g., P2P), and outline the techniques used to encrypt or even camouflage the malicious activities. We then use dynamic analysis traces to provide a long-term monitoring analysis on these 23 downloaders, identifying 18 downloaders to be still active as of writing this paper. In addition, we show that eleven downloader families are actively distributing malware for more than a year.

Motivated by this observation, we investigate how attackers ensure the resilience of downloader infrastructures. Contrary to our expectation that IP address blacklists would force attackers to change their infrastructure frequently, we show that 219 C&C servers (20%) were actively operated for more than four weeks. For the remaining servers, we analyze how attackers use DNS and IP address fluxing to operate their downloaders, suggesting that isolating downloader infrastructures is much harder than it seems.

As a third part of our analysis, we propose two automated methods to extract the downloaded malware (*eggs*) in a generic and scalable fashion. We hope that these techniques will support future efforts in analyzing downloaders without the manual effort of reverse engineering particular downloader families. We evaluate these two techniques both on downloaders with plaintext and encrypted communication, acquiring a diverse set of malware in the wild.

To summarize our contributions:

- We identify and characterize 23 malware downloaders, describing previously undocumented specimen and their communication models.
- We perform a long-term analysis of these downloaders, revealing that 11 downloaders have been operating for more than a year, and approach to understand the reasons for the infrastructural resilience.
- We propose two automated techniques to actively acquire malware from downloaders, without requiring reverse engineering downloaders.

## 2 Preliminaries

Malware defense mechanisms, especially anti-virus, have forced attackers to develop increasingly complex malware. This complexity has motivated attackers to specialize and separate duties. For example, services to stealthily install malware on computers may be provided by one group, while other fraudsters specialize in sending spam, and a third group could focus on keylogging. In this work, we focus on the service of installing new malware on systems via *downloaders*. Downloaders are malicious programs that

are instrumented to load additional malware via the Internet, which is in turn installed and executed on the victim's system.

### 2.1 Downloader Architectures

Figure 1 illustrates the architecture of a downloader. Once executed, a downloader contacts its command-and-control (C&C) server(s) via *C&C channels*. After receiving download instructions, it then establishes at least one *download channel* to load malware (*eggs*) via the network.

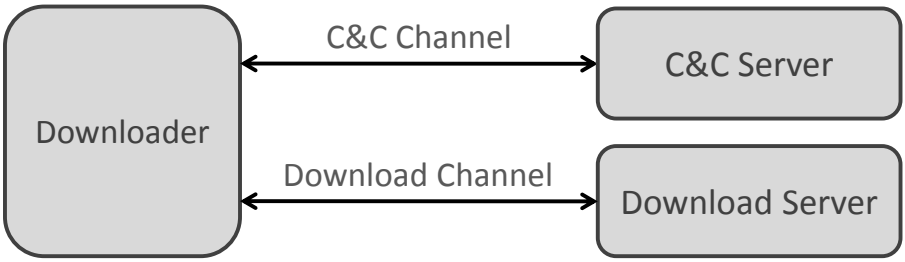



**Fig. 1.** Simplified architecture of a downloader: Seperation between C&C and download channel

**C&C Channels.** A downloader's C&C channel is used to get lists of URLs (or similar address information) where eggs can be downloaded from. Next to download instructions, the C&C channel can be used to report back to the C&C server if the download succeeded. In addition, as shown by Caballero et al. [5], C&C channels may exchange affiliate IDs in the economical model of pay-per-install downloaders. Moreover, downloaders send details about the infected host using the C&C channel, such as the OS version, username or device IDs. One characteristic of C&C channels is the *carrier protocol* used to transfer commands. Typical examples for carrier protocols are IRC, HTTP (e.g., if C&C messages are in the HTTP body) or plain TCP/UDP. The information exchanged on C&C channels is critical and highly subjective to counter-measures such as signature-based IDSs, and thus more advanced downloaders encrypt their C&C channel. During our investigations, we also observed downloaders that have download URLs hardcoded in their binaries. We excluded such downloaders from our analysis because of their simplicity and transitory nature.

**Download Channels.** We found the C&C and download channels to be typically well-separated. A download channel shares similar characteristics than the C&C channel, i.e., it has a specific carrier protocol and potential encryption schemes. While we did not see examples of steganographic C&C channels, as we will show, some downloaders tend to camouflage their malicious downloads in normal web traffic. Another typically distinguishing characteristic between C&C and download channels is the number of bytes transferred. C&C commands tend to be small, while eggs – no matter if encrypted or not – have significantly larger file sizes.

### 2.2 Related Work

First steps to analyze specific downloaders were made by Caballero et al. by analyzing four pay-per-install (PPI) programs [5]. PPI downloaders, a subset of downloaders in general, are based on an economical model cashing out attackers for installing malware on a freshly infected system. Caballero et al. implemented so called *milkers* to download eggs from these four PPI networks, and systematically analyze the ecosystem behind these networks. They show in-depth how egg families relate to download programs, and identified that kinds of malware (e.g., DDoS) were distributed in download campaigns.

Our work was inspired by Caballero et al., and we seek for a broader characterization of downloaders. In fact, we found ourselves at a position not knowing the magnitude and different types of downloaders currently active in the wild. We identify that the number and kinds of downloaders is significantly higher than expected. To the best of our knowledge, we are the first to approach a characterization of downloaders. We then also seek to answer the fundamental but yet unanswered question of how attackers build up infrastructures that are sufficiently resilient for long-term operations of downloaders. We expand a malware acquisition technique as proposed in Botlab [7] with replaying network dialogs as proposed by Newsome et al. [11]. While Botlab fetches malware from URLs found in spamfeeds, our techniques repeatedly acquires malware from downloader URLs. Existing systems like Threatexpert [15] or Anubis [4] can already analyze malware in general, but we are the first to analyze the behavior and infrastructures on downloaders over multiple executions and on long-term.

## 3 Analysis of the Downloader Landscape

In this section, we characterize and describe the 23 downloaders identified as part of this work, which we will then further analyze later in the paper.

### 3.1 Dataset Description

Our analyses are based on malware reports from Sandnet [12]. Sandnet executes and dynamically analyzes malware using Windows XP SP3 32bit virtual machines connected to the Internet via NAT. During malware execution, we deploy containment policies that redirect harmful traffic (e.g., spam, infections) to local honeypots. We further limit the number of concurrent connections and the network bandwidth to mitigate DoS activities. An in-path honeywall NIDS watched for security breaches during our experiments. Other protocols (e.g., IRC, DNS or HTTP) were allowed to enable C&C communication. We consider the biases affecting the following experiments due to containment to be negligible. Specifically, given our long measurement period of one hour per malware sample, we did not observe any incomplete download behaviors in our trace. Assuming that downloaders silently operate without the user's consent, we did not deploy user interaction during our experiments. We plan to analyze malware on 64-bit architectures or more recent Windows versions in the future.

Our dataset consists of 243,000 MD5 unique malware samples analyzed in Sandnet at least once between Feb 2010 and Feb 2012. We gratefully received these samples

from a variety of sources, including samples submitted to public dynamic analysis environments, feeds by security companies, our own honeypot infrastructures and spamtraps. While we cannot prove that this dataset covers all relevant malware families, it shows a diversity of 38,000 unique malware labels (according to Kaspersky). We extracted the malware family names from these labels and found over 1800 malware families in our dataset.

From this diverse set of samples, we scheduled a random selection on a daily basis, without giving any emphasis to particular malware families. To trigger the malware behavior, we then executed these samples for at least one hour. Obviously, only a minor fraction of these malware samples are in fact downloaders. To build up a dataset covering the most relevant downloaders we did a threefold approach. First, we consulted literature research and asked AV vendors for their expert knowledge on recent and prevalent downloaders. Second, in our dataset covering millions of malware samples, we searched for prevalent AV labels suggesting the malware is a downloader. Third, we manually inspected a random subset of the Sandnet analysis reports for downloader behavior. We manually filtered legitimate programs in our dataset of potential downloaders, such as e.g. Windows Update, Google Updater or programs to update system drivers.

For each identified downloader, we systematically searched for related analysis reports in Sandnet. Typically, we used payload or behavioral signatures to classify and recognize a particular downloader. In rare cases, where a downloader family did not expose any signature, we carefully assembled sets of domains and IP addresses to recognize downloader traffic. Using these techniques, we are able to detect all previous and upcoming executions of a particular downloader family.

### 3.2 Downloaders Overview

The resulting dataset provides an empirical overview of existing downloaders. Table 1 lists the downloaders that we monitor as part of this work. While this is not necessarily complete, it shows a large diversity in terms of different downloader characteristics. The attributes in Table 1 form two groups: The left-hand attributes characterize the C&C channel, while the right-hand columns characterize the download channel. We labeled three droppers with generic names (dldr-#1 to dldr-#3), as anti-virus vendors either assigned too generic or contradictory labels for those.

**Carrier Protocols.** A first distinction between the downloaders can be made in terms of the *carrier protocol*, that is, the protocol used to communicate with C&C or Download servers. To understand and also classify downloaders, we had to reassemble and parse numerous carrier protocols (UDP, TCP, DNS, HTTP, IRC, TLS). For obfuscated protocols, we define the carrier protocol to be the underlying protocol of the C&C protocol, e.g., “HTTP” for GoldInstall or Renos/Artro and “TCP” for the encrypted variant of Virut C&C. Interestingly, Table 1 shows that C&C channels are not necessarily designed in the same way as download channels. For example, five downloaders use obfuscated or encrypted C&C channels, but at the same time have plaintext HTTP download channels. Similarly, another five downloaders do not separate between C&C and download channels, abbreviated by “inl” to show that malware is served inline with the C&C protocol.

**Table 1.** Overview of downloaders under our analysis. Columns 2–5 characterize the C&C channel, columns 6–9 characterize the download channel. "Pl?" shows if the communication channel was in plain text, and "DNS?" shows if names of the communication endpoints were resolved via DNS prior to contacting them.

| | *C&C Channel* | | | | *Download Channel* | | |
|---|---|---|---|---|---|---|---|
| *Family* | *arch* | *Pl?* | *Protocol* | *DNS?* | *Pl?* | *Protocol* | *DNS?* |
| Renos/Artro | cent | ✗ | HTTP | ✓ | ✗ | HTTP | ✓ |
| Sality | cent | ✗ | HTTP | ✓ | ✗ | HTTP | ✗ |
| dldr-#1 | cent | ✗ | HTTP | ✓ | ✗ | HTTP | ✓ |
| Cycbot/Gbot | cent | ✗ | HTTP | ✓ | ✗ | HTTP-inl | ✓ |
| Karagany | cent | ✗ | HTTP | ✓ | ✗ | HTTP-inl | ✓ |
| Gamarue | cent | ✗ | HTTP | ✓ | ✓ | HTTP-inl | ✓ |
| Dofoil | cent | ✗ | HTTP | ✓ | ✓ | HTTP | ✓ |
| Emit | cent | ✗ | HTTP | ✓ | ✓ | HTTP | ✓ |
| GoldInstall | cent | ✗ | HTTP | ✓ | ✓ | HTTP | ✓ |
| Rodecap | cent | ✗ | HTTP | ✓ | ✓ | HTTP | ✓ |
| Virut (crypt C&C) | cent | ✗ | TCP | ✓ | ✓ | HTTP | ✓ |
| TDSS | cent | ✗ | TLS | ✓ | ✗ | HTTP | ✓ |
| Winwebsec | cent | ✓ | HTTP | ✗ | ✓ | HTTP | ✗ |
| Dabvegi | cent | ✓ | HTTP | ✓ | ✗ | HTTP | ✓ |
| Buzus | cent | ✓ | HTTP | ✓ | ✗ | HTTP | ✓ |
| dldr-#3 | cent | ✓ | HTTP | ✓ | ✓ | HTTP | ✓ |
| Zwangi | cent | ✓ | HTTP | ✓ | ✓ | HTTP | ✓ |
| Harnig/LoaderAdv | cent | ✓ | HTTP | ✓ | ✓ | HTTP-inl | ✓ |
| dldr-#2 | cent | ✓ | HTTP | ✓ | ✓ | HTTP-inl | ✓ |
| Virut (plain C&C) | cent | ✓ | IRC | ✓ | ✓ | HTTP | ✓ |
| Vobfus/Changeup | cent | ✓ | TCP | ✓ | ✓ | HTTP | ✓ |
| Sality P2P | P2P | ✗ | UDP | ✗ | ✗ | TCP | ✗ |
| Zeus P2P | P2P | ✗ | UDP | ✗ | ✗ | TCP | ✗ |

**Communication Architectures.** As Table 1 suggests, almost all downloaders deploy a centralized C&C architecture. Two exceptions are Sality P2P and Zeus P2P. Sality uses a hybrid C&C architecture, i.e., some samples use a centralized HTTP-based C&C channel while others receive their commands via a peer-to-peer network. Zeus P2P is a pure P2P based bot with download functionality. Such distributed networks are attractive to attackers, as the C&C infrastructure cannot be disrupted by taking offline single C&C servers. Both Sality P2P and Zeus P2P[1] initialize their C&C channel by trying to contact hundreds of P2P bootstrapping nodes.

**DNS.** Table 1 reveals that most downloaders make use of DNS to resolve the names of their C&C and/or download servers. However, downloader families such as Winwebsec and the P2P-driven downloaders avoid DNS resolution for both C&C and download servers. We speculate that such downloaders either have no technical need for DNS, e.g. the P2P architectures, or want to foil malware domain blacklists. From the attacker's point of view, another disadvantage of using DNS is that taking down domains exposes an additional point of failure in the communication chain. However, on the other hand,

[1] Note that while Zeus may not be conceived as downloader, there are references supporting our observation that recent Zeus variants drop other malware.

DNS would allow to quickly redirect to different IP addresses of download servers. This dilemma basically boils down to: Who is more resilient, the hoster (IP) or the DNS provider (domain)? We try to shed light onto different resilience strategies in Section 4. The fact that most downloaders use DNS resolution shows that developing mitigation techniques based on DNS is promising. Howevr, although we see downloaders using DNS, they may also have a backup communication channel, e.g., using hardcoded IP addresses [10].

Intuitively, one may think that downloaders use DNS to quickly react on server takedowns. Fast flux [9], domain flux and the business of bullet-proof DNS hosting would support this intuition. As we figured, however, some downloaders do not (need to) change DNS records of particular C&C domains. Consequently, while the usage of domains evolved over time, the IP addresses resolved by these domains were relatively static. We will further analyze these observations in Section 4.1.

**Communication Encryption.** Defense mechanisms such as network-based intrusion detection systems or anti-virus scanners scan for URLs and file contents downloaded from the Internet. As a consequence, downloaders deploy a wide set of schemes to obfuscate or encrypt their communication channels. A distinction can be made between deploying well-known or custom encryption techniques. For example, the TDSS downloader relies on TLS within its C&C channel, thus preventing from eavesdropping on C&C communication [13]. Similarly, we observed that the Renos/Artro family encrypts using RC4 with a key hardcoded in the samples.

In contrast, other downloaders use custom encryption/obfuscation algorithms. To give insights, we reverse engineered specific downloader families. For example, Emit deploys an XOR shifting technique to obfuscate traffic. Similarly, Virut picks a random session key and the C&C servers derive these session keys by performing a known-plaintext attack on the ciphertext of the first message sent from the bot to the server. The session key itself is thus never transmitted. Independent from the cryptographic strength of a particular algorithm, understanding and possibly decrypting the ciphertexts often requires tremendous reverse engineering efforts.

**Steganography.** Attackers further disguise the egg downloads with steganography. While encryption prevents eavesdroppers to read exchanged data, steganography tries to hide the existence of egg downloads. We have spotted camouflage techniques used by downloaders that could be interpreted as first steps towards steganography. For example, Renos/Artro hides its eggs in valid GIF files. Although these files look like regular legitimate pictures, eggs are carried as part of the files. Using custom routines, the downloader transforms these files to correct PE binaries.

**Downloaders Using Public Services.** Most downloaders rely on their own infrastructure for hosting malicious software. However, we also observed that particular downloaders make use of publicly accessible services. For example, dldr-#1 retrieves its malicious files from a large public file clouding provider. From a defender's perspective, it is much harder to block access to legitimate services, as a distinction between legitimate or malicious downloads from such sources raises big challenges.

**Tracking Mechanisms.** Among the plaintext downloaders, we could observe downloaders that are client-aware. That is, attackers derive pseudo-unique IDs per system,

such as e.g. its MAC address, the gateway's public IP address or the Windows serial. The C&C servers can then keep track of which clients contacted them, and serve binaries accordingly. Similarly to e.g. Torpic [14], the downloader victims are presumable identified to track the number of infections, either to keep an overview or to use this data for payment (e.g., PPI). Another reason would be to observe and defend against potential abuses of the downloader infrastructures (see Section 5). To work around this in our setup, we are modifying fixed strings such as the MAC address for every malware execution in Sandnet since ever.

### 3.3 Downloader Lifetime

With our understanding that downloaders are a fundamental part of the malware lifecycle, we now analyze the lifetimes of the downloaders. For this lifetime analysis, we are not interested in a particular downloader *binary* (identified by the MD5 hash sum). Instead, we analyze when a particular downloader *family* appears in our dataset, and how long its C&C or download activities continue.

As a first step, we used the mechanisms described in Section 3.1 to identify downloaders of a particular family in our dataset. We specifically designed our signatures to match evolutions of particular downloaders. For example, GoldInstall, a PPI program with diverse affiliation programs [5], was covered by a single signature. We then had to filter C&C flows that reached the C&C server, but the C&C server responded with non-C&C data (e.g., HTTP 404 responses). We enhanced our signatures with heuristics verifying that an endpoint shows active C&C communication, filtering out a significant amount of sinkholed communication.

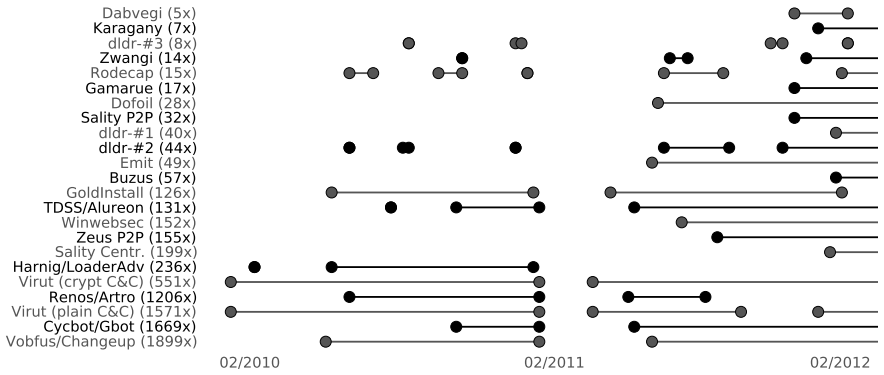



**Fig. 2.** Lifetime of downloaders, as observed in Sandnet, from Feb 2010 until Feb 2012. The numbers in brackets represent the number of active executions of this downloader in Sandnet.

Figure 2 shows the resulting activity plot. To increase readability, we connected two markers if the gap between these two downloader occurrences in our dataset was less than four weeks. Due to a maintenance period in Sandnet, the graph lacks activity measures of droppers between 03/02/2011 and 08/04/2011. Overall, however, the graph

shows that at least 11 of the 23 downloaders (48%) actively operated for more than a year. In addition, 18 downloaders (78%) are still active as of writing this paper. Given that some downloaders are more present in our sample feeds than others, and given that our measurement period started in Feb 2010, the resulting data represents lower bounds of the actual dropper lifetimes. We even noticed that some downloader families were discussed by the community prior to our measurement period, indicating that the lifetimes of some downloaders is significantly longer than two years. We therefore speculate that in fact even more downloaders were successfully operated in long-term. This opposes a long-lasting threat to our community, as apparently downloaders are largely and continuously used to infect PCs.

A few downloaders, such as Dofoil or Gamarue appeared first in our dataset in 2011, underlining active developments in the malware scene. The reasons why other downloaders ceased during the measurement period are twofold. First, in case of GoldInstall, C&C servers were not responsive for weeks, potentially indicating a downloader was abandoned or undergoes a major evolution. Second, as of August 2011, all specimen of Renos/Artro in our dataset were sinkholed by Shadowserver or Spamhaus.

## 4 Downloader Infrastructures

Seeing the significant lifetimes of downloaders, and knowing that defenders try to mitigate the threats of malware in general, we asked ourselves: How, technically, do attackers ensure such a high and long-term availability of their downloader infrastructures? In this section, we will therefore investigate the critical infrastructures used by attackers to operate their downloaders, i.e., both C&C and download servers.

### 4.1 C&C Infrastructure

C&C servers are vital to instrument the downloaders with new download instructions, and thus represent a sensitive part in the architecture of downloaders. From a downloader's perspective, two infrastructural services are crucial. First, most downloaders depend on DNS resolution prior to contacting their C&C server. Second, C&C servers obviously need to be reachable and service correctly. From a defender's perspective, both hosts (IP addresses) and domains represent vantage points to detect and/or disrupt downloaders.

We use the data obtained in Section 3.3 for further analyzing the C&C infrastructure. In particular, we aggregate the number of domains and IP addresses used by a particular downloader as observed in Sandnet. While this does not necessarily give a complete view on the IP addresses and domains used by a downloader, the numbers can serve as lower bounds. Table 2 shows that the resilience strategies differ between the downloaders. In the second major column, we summarize statistics on the specific C&C server IP addresses of a downloader, plus its Autonomous System (AS). In the third major column, Table 2 lists the number of C&C domains per dropper. We highlighted domains or IP addresses that we have seen in active use for at least four consecutive weeks in Table 2 in the columns annotated with "LL" (long lasting).

**Table 2.** Statistics on the C&C server distribution infrastructure per downloader family. LL=Long Lasting, i.e., IP addresses/domains had an uptime of more than 4 weeks. In each such case, we increase the corresponding AS/TLD counter by one also.

| *Downloader* | *IPs* | | *ASes* | | *Domains* | | *TLDs* | | *Timespan* |
|---|---|---|---|---|---|---|---|---|---|
| *Family* | *#* | *# LL* | *#* | *# LL* | *#* | *#LL* | *#* | *#LL* | M/Y - M/Y |
| Buzus | 2 | 1 | 1 | 1 | 3 | 2 | 2 | 1 | 01/12 - 02/12 |
| Cycbot/Gbot | 145 | 48 | 56 | 36 | 2347 | 57 | 6 | 6 | 10/10 - 02/12 |
| Dabvegi | 5 | 4 | 4 | 3 | 5 | 4 | 3 | 3 | 11/11 - 01/12 |
| dldr-#1 | 69 | 19 | 4 | 2 | 5 | 2 | 3 | 2 | 01/12 - 02/12 |
| dldr-#2 | 41 | 11 | 21 | 5 | 45 | 12 | 7 | 4 | 06/10 - 02/12 |
| dldr-#3 | 10 | 1 | 2 | 1 | 10 | 2 | 4 | 2 | 08/10 - 01/12 |
| Dofoil | 12 | 2 | 7 | 2 | 16 | 0 | 3 | 0 | 06/11 - 02/12 |
| Emit | 7 | 2 | 2 | 1 | 9 | 4 | 1 | 1 | 06/11 - 02/12 |
| Gamarue | 80 | 3 | 57 | 3 | 12 | 1 | 4 | 1 | 11/11 - 02/12 |
| GoldInstall | 12 | 5 | 7 | 3 | 13 | 8 | 3 | 2 | 05/10 - 01/12 |
| Harnig/LoaderAdv | 24 | 11 | 6 | 1 | 42 | 32 | 1 | 1 | 03/10 - 01/11 |
| Karagany | 2 | 0 | 1 | 0 | 7 | 0 | 2 | 0 | 12/11 - 02/12 |
| Renos/Artro | 27 | 5 | 12 | 3 | 75 | 0 | 3 | 0 | 06/10 - 02/12 |
| Rodecap | 8 | 4 | 2 | 2 | 5 | 4 | 3 | 3 | 06/10 - 02/12 |
| Sality Centr. | 239 | 62 | 125 | 47 | 243 | 59 | 31 | 18 | 06/11 - 02/12 |
| Sality P2P | 9849 | 1457 | 900 | 424 | 0 | 0 | 0 | 0 | 11/11 - 02/12 |
| TDSS/Alureon | 28 | 8 | 21 | 8 | 28 | 3 | 1 | 1 | 08/10 - 02/12 |
| Virut (crypt C&C) | 20 | 6 | 11 | 6 | 44 | 10 | 3 | 2 | 02/10 - 02/12 |
| Virut (plain C&C) | 14 | 4 | 9 | 4 | 3 | 3 | 1 | 1 | 02/10 - 02/12 |
| Vobfus/Changeup | 19 | 8 | 14 | 7 | 17 | 13 | 3 | 3 | 05/10 - 02/12 |
| Winwebsec | 5 | 2 | 4 | 2 | 0 | 0 | 0 | 0 | 10/10 - 02/12 |
| Zeus P2P | 2140 | 31 | 446 | 21 | 0 | 0 | 0 | 0 | 08/11 - 02/12 |
| Zwangi | 97 | 7 | 4 | 1 | 10 | 1 | 1 | 1 | 10/10 - 02/12 |

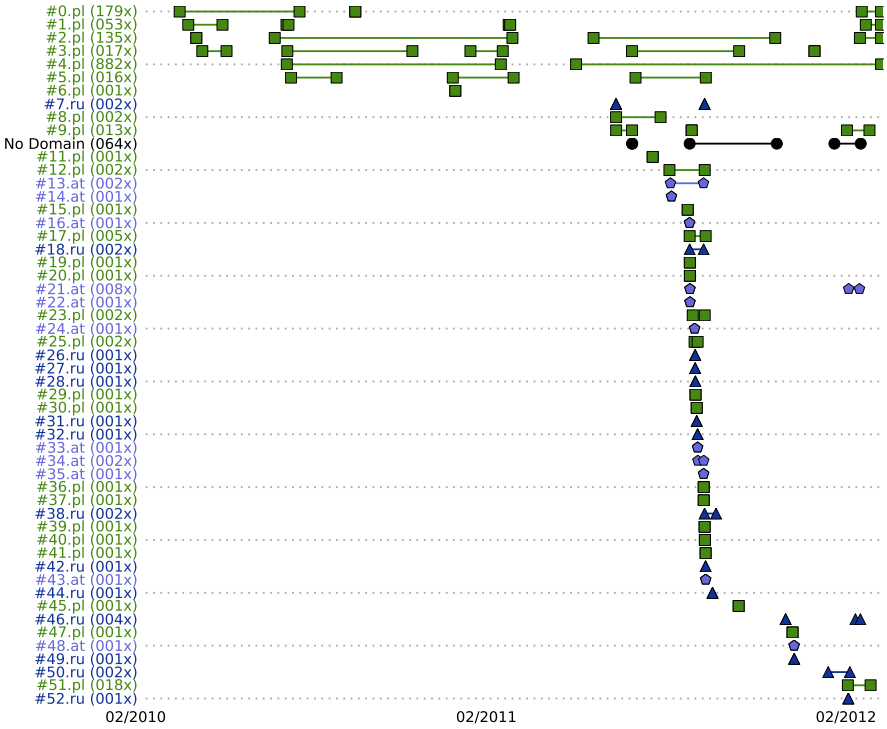
Table 2 reveals that most downloaders use multiple C&C server hosts, and tend to distribute their servers across network boundaries. For example, Virut has been in operation during our entire analysis period with about 20 IP addresses in eleven ASes. We speculate that spreading server locations among multiple ASes is a strategic decision by the attackers. The more responsible parties and different national regulations are in place, the higher the complexity for defenders to take actions against specific downloaders. In that sense, Cycbot/Gbot stands out with 146 servers, hosted in more than 50 different networks. Observing such a large diversity may indicate that Cycbot/Gbot is in fact a malware toolkit with downloader functionality, which results in many smaller infrastructures independent from each other. We verified that the Cycbot/Gbot instances in our dataset used different IP addresses at approximately the same time. Another interesting case is dldr-#1, which appears to operate many C&C servers on its own. But instead it uses a large public file sharing company and this hoster's load balancing techniques, hiding eggs in seemingly benign Bitmap image files. As we are interested in all C&C activities of a downloader, we manually inspected all cases where possibly benign IP addresses or domains (e.g., image hoster) were involved and we explicitly did not exclude them from Table 2 if we also detected C&C.

Outstanding are the P2P variants of Zeus and Sality, with more than thousands of different C&C "server" hosts each. For these downloaders, we consider P2P neighbors that respond to P2P-related UDP requests as active. The large number of ASes involved, 900 for Sality P2P and 446 for Zeus P2P, show that provider-driven initiatives against these P2P networks are deemed to fail. Interestingly, and particularly for Sality P2P, we saw a large fraction of P2P nodes to be lasting for more than four weeks. We first thought this may indicate that defenders joined this particular P2P network, but the high number of long-lasting ASes speaks against this.

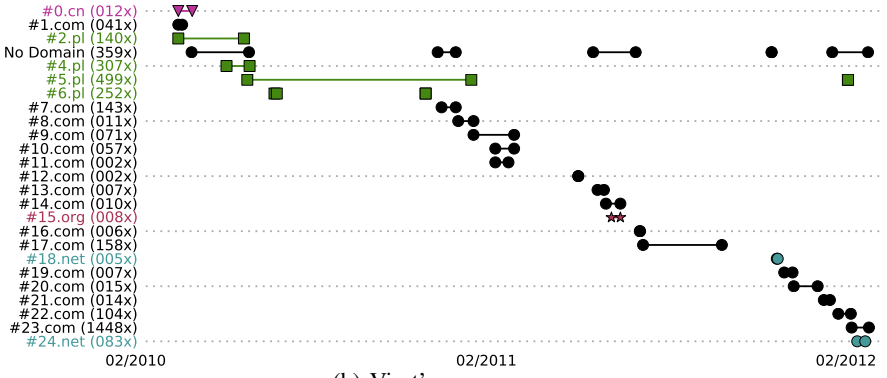
The analysis on the domains used by downloaders provides further interesting insights. Zwangi, for example, heavily rotates its C&C IP addresses typically within four /22 networks. Similarly, Gamarue deploys one particular domain pointing to highly fluctuating IP addresses in over 50 different ASes. In both cases the IP addresses are typically reused, i.e., DNS is used to steer downloaders towards multiple C&C servers. On the other hand, we observed downloaders for which the set of IP addresses was relatively constant, but the domains to resolve these IP addresses changed over time. For example, Virut used 45 domains to resolve to its 20 C&C servers, and Renos/Artro pointed its 136 domains to 38 IP addresses. Related to the previous observation that attackers settle their C&C servers in multiple networks, we also show that – for most downloaders – a diverse set of Top Level Domains (TLDs) is chosen. Usually, these C&C domains are even registered across many continents, mostly including European, South-/North-American, and Asian registrars. Again, involving multiple domain registrars is presumably a strategic decision.

It can be seen that a large fraction of C&C servers (20%) remains operable for more than four weeks. Similarly, 217 domains pointing to active C&C servers (7%) remain in active use for at least four weeks. The observed long-levity enables defenders to take actions against downloaders, such as using domain or IP address blacklists. On the other hand, the involvement of numerous registrars and providers shows how complex takedown efforts would be.

As a case study, we compared the usage time spans of Virut's C&C server domains (Figure 3(a)) with the usage time spans of the egg download server domains (Figure 3(b)). Both figures reveal that Virut seems to have a subset of stable domains that have been used throughout the last two years and that are still in active use, for both C&C and egg servers. In addition, several domains have been used only for certain periods. However, the sets of domains for C&C and egg distribution are distinct, i.e. we have not witnessed a single domain being used for both, C&C and egg distribution. Interestingly, we observed a churn of Virut C&C server domain names between June 2011 and January 2012. Our initial hypothesis that these domains were used as backup C&C domains was proven wrong, as many other domains have been actively in use during that period. In addition, our passive DNS database in Sandnet revealed that not a single DNS resolution request for a Virut C&C domain resulted in NXDOMAIN or an empty answer section – Virut thus exhibits a remarkable C&C and egg server availability. We leave a more fine-grained analysis to measure to which extent C&C servers/domains are responsive simultaneously to future work.




(a) Virut's C&C server domains




(b) Virut's egg servers

**Fig. 3.** Virut's C&C (above) and egg (bottom) server usage by domain over time. Colors/markers denote top level domains.

### 4.2 Download Server Infrastructure

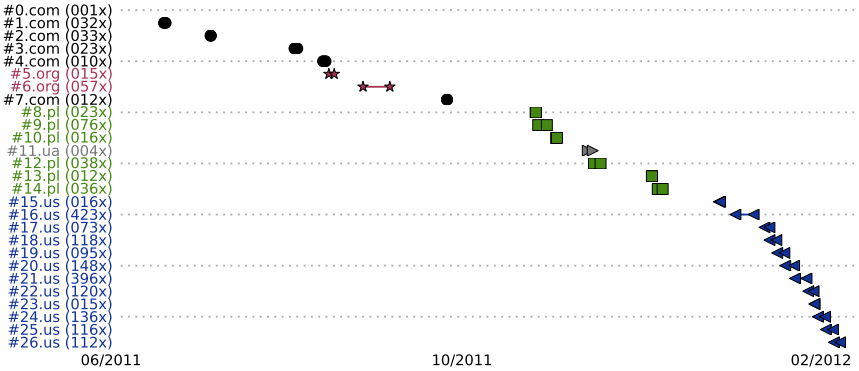
The second pillar of a downloader's infrastructure are the download servers. We will now analyze the infrastructures of downloaders with plaintext download channels. We focus on plaintext downloaders, as we could map download channels to downloaders in these cases with a high accuracy. Table 3 shows statistics on the egg distribution infrastructure for these downloader families. On purpose, we do not consider the C&C infrastructure here, except – unavoidably – in cases where the egg sample download is part of the C&C channel.

**Table 3.** Statistics on the egg sample distribution infrastructure per downloader family. LL=Long Lasting, i.e., uptime of more than 4 weeks. Packers: u=UPX, t=Themida, p=PECompact, e=PEtite, b=BobPack/Bobsoft, a=Armadillo, s=ASPack/ASProtect, x=EXECryptor, h=Thinstall, n=NsPack, f=FSG, d=D1S1G, v=Upack, c=CrypKey, o=ProActivate, y=XtremeProtector, w=WinUpack, N=NET MS, M=MoleBox, Y=y0dasCrypter.

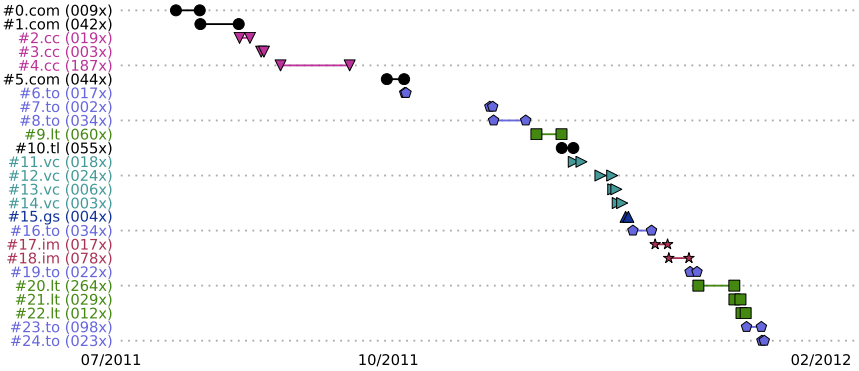
| *Downloader* | *IPs* | | *Domains* | | *Eggs* | | *Maximum* | *Packers* | |
|---|---|---|---|---|---|---|---|---|---|
| *Family* | *#* | *#LL* | *#* | *#LL* | *#* | *#MD5s* | *Uptime* | *#* | *Detected* |
| dldr-#2 | 26 | 7 | 44 | 10 | 1029 | 110 | 561 days | 6 | b,t,u,f,h,y |
| dldr-#3 | 8 | 1 | 9 | 1 | 648 | 158 | 114 days | 7 | u,s,d,n,p,c,e |
| Dofoil | 14 | 1 | 29 | 0 | 103 | 93 | 96 days | 3 | u,c,Y |
| Emit | 6 | 2 | 27 | 0 | 5938 | 698 | 183 days | 2 | u,p |
| GoldInstall | 70 | 25 | 63 | 16 | 13155 | 971 | 592 days | 8 | u,b,s,v,p,w,n,N |
| Harnig/LoaderAdv | 31 | 12 | 46 | 23 | 1731 | 735 | 185 days | 9 | u,o,f,p,d,N,b,n,M |
| Rodecap | 2 | 2 | 8 | 2 | 286 | 23 | 445 days | 2 | u,a |
| Virut (crypt C&C) | 30 | 8 | 25 | 6 | 3852 | 293 | 459 days | 5 | u,x,n,s,d |
| Vobfus/Changeup | 15 | 7 | 34 | 3 | 2005 | 424 | 77 days | 1 | u |
| Winwebsec | 6 | 1 | 1 | 1 | 80 | 22 | 58 days | n/a | ukn |
| Zwangi | 86 | 2 | 8 | 1 | 263 | 138 | 49 days | n/a | ukn |

Two thirds of the observed plaintext downloader families exhibit more than ten distinct IP addresses for their sample servers. A similar trend is observed concerning the domain names – only the Winwebsec family does not make use of DNS at all in the egg download process.

Per downloader family, the maximum uptime expresses the maximum time span where one single egg server IP address has been witnessed as serving egg samples. Note that, in comparison to Table 2, the measurement in Table 3 is restricted to a family's egg servers and omits its C&C infrastructure. We consider an IP address or domain as long lasting if it serves eggs for at least four weeks. Table 3 shows that over the whole monitoring period, only a small fraction of the IP addresses is actually long lasting. In the cases that we manually inspected, we observed that downloaders typically move their download servers from time to time. For each downloader family of Table 3, we manually inspected the egg server usage over time for both, domains and IP addresses. Interestingly, all downloader families exhibit similar egg server usage patterns where the migration from one domain to another is clearly visible. The same applies to the IP addresses of egg servers, however, egg server domains typically change more often




(a) Emit: Gradually migrating to new domains and sometimes new TLDs




(b) Vobfus/Changeup: Migrating to new, previously unseen TLDs

**Fig. 4.** Egg server usage by domain over time for two downloaders. Domain names have been pseudonymized. Marker styles and colors distinguish the download server's top level domain.

than IP addresses. Some of the servers that we observed to be long lasting, even actively serve eggs for more than a year.

For the downloader family Emit, Figure 4(a) shows each egg server domain on the y-axis and the associated usage time spans. Note that the domain names have been pseudonymized. The egg server domains show hardly any overlap in their usage time spans. In addition to the usage time span, the marker and the color denote the top level domain. We observe that not only does the egg server move from one domain to another – indicated by the pseudonym – it also migrates from one top level domain to another, i.e. from initially .com to .org, .pl and finally to .us. This pattern shows that – in order to strive for a takedown of this downloader's egg serving infrastructure on the DNS level – many different registrars would be required to cooperate.

Figure 4(b) Vobfus/Changeup, which exhibits a strong domain migration pattern for its egg servers. In this case, the domain names are typically only used for a couple

of days, and never reused. Not as consistent as Emit, but still, Vobfus exhibits sequential top level domain migration, too, although a few top level domains are used in parallel.

## 5 Egg Acquisition and Analysis

After investigating the downloader infrastructure, we will now analyze the downloaded eggs. Such an analysis allows us to draw conclusions on how attackers operate the egg infrastructure, e.g., by using polymorphism and aggressively repacking served samples. We will begin with presenting two techniques how to acquire eggs from both plaintext and encrypted downloaders. The resulting dataset of actively acquired eggs will then serve to give first insights into evasive techniques used by downloaders.

### 5.1 Egg Acquisition Techniques

All downloader infrastructures have one necessity in common: these services must be publicly accessible, as (with the exception of targeted attacks) fraudsters aim for large-scale deployment of their malware. Consequently, attackers cannot easily deploy client authentication mechanisms that prevent their infrastructures from being "abused", raising the difficulty for attackers to control who is accessing the infrastructures. We exploit these necessities to obtain eggs for the downloaders under our analysis. We present two techniques that enable us to acquire the downloaded eggs for plaintext and encrypted droppers. Previous efforts analyzing a few specific downloaders [5] did not require automated egg acquisition techniques. However, given our significantly larger sample set, we seek for a more scalable solution to analyze downloaders. Our techniques may be a potential enabler for future research on malware acquisition methods or egg analysis, as our methods do only require little manual effort compared with reverse engineering.

**Plaintext Downloaders.** For plaintext downloaders, we exploit the fact that eggs are downloaded without disguising or encrypting the communication. Methodically, we replay the egg-download dialog towards each download server and require new egg samples this way. For example, in case of HTTP, once the download server and the egg's URI is known to defenders, downloads can be repeated regularly. We implemented a *dialog repeater* that takes pairs of HTTP request and communication endpoint as input, i.e., payload bytes with a destination IP address and port. For each such pair, the repeater replays the dialog towards the specified destination once an hour, typically resulting in HTTP responses. We feed the repeater with input pairs by searching for requests by downloaders in our dataset that led to egg downloads. Given the prevalence of HTTP in our dataset, we left it open for future work to incorporate further network protocols to the dialog repeater.

In order to avoid such mechanisms, fraudsters could potentially use blacklists of IP addresses of known malware analysis systems [1]. For an attacker, it is straightforward to block all requests from systems as ours. Consequently, instead of using a single Internet outbreak and IP address, we established a proxy network to route the traffic through our home DSL lines. In contrast to well-known proxies such as Tor or open proxies, end-user IP addresses seem to stem from realistic end-users and – in our case – even change daily. We made sure that our ISPs did to interfere with our measurements

by comparing outputs of multiple proxy hosts. Despite its simplicity, as we will show, the repeater is a well-working mechanism to acquire new eggs.

**Encrypted Downloads.** A drawback of the dialog repeater is that it cannot milk eggs from downloaders using encrypted download channels. Even if the download succeeded, we could not make use of the encrypted egg. Therefore, as a complementary technique, we leverage the actual downloader to acquire eggs. The intuition behind this method is simple: whenever a downloader is executed, it will download and execute previously unknown malware samples. We instrumented our Sandnet VMs with a kernel-based Windows system driver that records the file images whenever new processes are forked or system drivers are loaded. For each potential egg being executed, the kernel driver computes the MD5 checksum and records the new processes' image.

However, monitoring new processes results in a large amount of legitimate system files to be interpreted as potential egg. To filter legitimate system files, we built a whitelist of trusted system files by scanning all files of a clean Sandnet VM. In addition, as a further filter to catch only actually dropped and not modified system files, we manually assembled patterns for the file paths where each downloader is storing its eggs. We specifically discard eggs that we identify as exact or repacked/modified copies of the downloaded itself by correlating the time when data was received from the network with the time when the new process was forked. After adding the kernel driver to Sandnet, we additionally scheduled the downloader families with encrypted download channels for execution in Sandnet on a daily basis for seven weeks starting in Jan 2012.

### 5.2 Egg Sample Distribution

In addition to our passive Sandnet database, we use both active techniques described to obtain a comprehensive egg dataset. Thus, for the plaintext downloaders, we identified the download channels and additionally describe the downloader infrastructure and their uptime. Table 3 (page 54) shows the egg distribution per downloader family that exhibit plaintext egg downloads.

The number of successful egg downloads as well as the number of MD5 unique egg samples differs widely among the plaintext downloader families. Whereas for GoldInstall more than 13,000 egg downloads completed successfully, the number of unique egg samples is much smaller. Other families such as Dofoil show that still a significant fraction of the successful egg downloads expose differing MD5s.

Table 4 summarizes our experiments of actively milking encrypted downloaders in Sandnet. For each downloader, we name the number of executions in Sandnet and show the number of eggs and unique eggs, respectively. For nine of ten downloaders, our technique was able to trace eggs. Despite its short runtime and the relatively small number of execution per downloader, we were able to acquire a high diversity of eggs. For example, although Zeus is well-known for keylogging and information stealing, we can confirm Symantec's recent observation [3] that it also downloads non-Zeus samples. For Sality P2P, we have observed active downloads, but the eggs were never executed during our monitoring period. Consequently, our kernel driver did not record new processes. Renos/Artro drops malware, although from August 2011 on our Renos samples were effectively sinkholed. Independent from the fact that all eggs were included in

**Table 4.** Downloaded egg samples from the encrypted downloaders from Dec '11 to Feb '12. Packers: u=UPX, t=Themida, p=PECompact, b=BobPack/Bobsoft, c=CrypKey, z=StealthPE.

| *Family* | *Execs* | *Eggs* | *MD5s* | *Packers* |
|---|---|---|---|---|
| Buzus | 316 | 1898 | 329 | b,u,p |
| Cycbot/Gbot | 181 | 1030 | 374 | u,c |
| Dabvegi | 278 | 271 | 8 | unknown |
| dldr-#1 | 14 | 10 | 3 | t |
| Karagany | 256 | 242 | 178 | z |
| Renos/Artro | 320 | 2454 | 23 | u |
| Sality | 261 | 241 | 59 | u |
| Sality P2P | 250 | 0 | 0 | n/a |
| TDSS/Alureon | 226 | 652 | 79 | n/a |
| Zeus P2P | 224 | 221 | 101 | n/a |

the original downloader sample, our technique could in fact extract the eggs. Both for Renos/Artro and Sality P2P, we could in general milk the downloaders, if we executed more recent samples or increased the analysis period. The low number of executions of dldr-#1 is due to scheduling this downloader only recently. As a particularly interesting case, TDSS/Alureon dropped all recorded executables by extracting the original sample. In addition to the loaded eggs, however, we recorded that in about half of the executions a kernel driver was loaded, showing that our technique may even work for downloaders with rootkits capability.

### 5.3 Polymorphism

Malware is well-known for polymorphism in order to evade antivirus signatures. An interesting question in the context of downloaders is whether and how polymorphic code is used. We approached this aspect twofold. First, we classified all egg samples using yara [2] and packer identification rules in order to assign which packer was used to (re)pack an egg sample. In addition, we submitted the egg samples to our sample sharing partners and Virustotal. In turn, querying Virustotal, we were thus able to assign A/V labels to the egg samples.

**Sample Packing.** A large fraction of the egg samples were successfully classified using yara packer rules. Tables 3 and 4 show the number of distinct packers for the eggs of each downloader family. The dominating packers are based on UPX. However, many different packers can be found, such as Armadillo, Themida, ASPack, ASProtect, NsPack and PECompact. In addition, some eggs, such as those of Winwebsec, were packed with unknown packers. The fact that the egg packers vary throughout one downloader family, supports the assumption that there are multiple "clients" per downloader and that it is likely not the download server that repacks the eggs. Instead, we assume that the clients make packed eggs available to the downloaders. In this context, we consider a client to be an attacker willing to distribute malware via downloaders.

**Repacking.** In order to successfully evade signature-based A/V, eggs are repacked at certain intervals. For those families that have plaintext egg downloads, based on Sandnet and dialog repeater traces, we estimate lower bounds on which downloader families distribute polymorphic eggs. In this context, we define an egg to be repacked if different content – in terms of MD5 hash – is served for what can be considered the same egg sample – based on (approximate) file size and A/V label. Thus, for each downloader family, we consider an egg to be repacked if we observe egg downloads with at least 8 distinct MD5 egg hashes all having (nearly) the same file size (rounded to kilobytes) and the same A/V label, within a time span of one month. On average, our filter criteria translate to a repacked egg sample at least once every four days. Furthermore, to ensure statistical significance, we limit our dataset for this experiment to families with at least 90 distinct eggs. Of those 9 families, we observed 8 to exhibit repacked samples. Note that we do not consider repacking to be a property of the downloader family. Instead, we assume that the clients of these downloaders take care of the repacking of their eggs. Wheres at least one client of Emit reached a maximum repacking rate of once every 17 minutes, dldr-#3 only repacked up to once every 2.5 days. For GoldInstall, we measured repacking once a day, and one of the Dofoil clients repacked its eggs once every hour.

This confirms similar analyses by Cabellero et al. [5], only that two downloaders in our dataset (Emit, Dofoil) deploy overly aggressive repacking. Employing our dialog repeater, we looked for server-side polymorphism where the egg sample is repacked upon *each* request. In particular, we tried to measure whether the repacking of Emit eggs takes place via on-the-fly server-side polymorphism, but unfortunately the egg servers have not been reachable during this experiment.

## 6 Discussion and Future Work

*Motivation of this work:* Our analyses provide detailed, novel and important insights into malware downloaders, but one may wonder if revealing such data has positive effects to the security community. In particular, revealing the possibility to monitor downloaders may motivate attackers to switch to more advanced techniques. However, given the large numbers of long-term operating downloaders, we see the need to raise attention to this problem domain. Our work also aims to highlight relevant downloader families, as e.g. P2P- or rootkit-driven downloaders, fostering future research on potentially previously unknown malware families.

*Evasion:* Obviously, our techniques to automatically milk downloaders are evadable by attackers. While it is straightforward to evade our dialog repeater, evading our kernel-based driver requires more thoughts. For example, we face the risk that our current setup may fail for kernel-level rootkits such as TDSS. Similarly, we had to exclude one particular downloader (Wintrim) from our analysis, as it detects virtualized environments. However, hardened dynamic analysis as with Ether [6], hardware-based hosts [8], or developing resilient kernel drivers would be effective against attackers' moves.

*Containment:* During dynamic analysis, and particularly when allowing network access to malware, we potentially risk to harm others. However, in a best effort to drop all harmful traffic, we strictly control and monitor Sandnet's activity. As a consequence, we have not observed a single abuse complaint concerning Sandnet, so far. Furthermore,

a particular risk of executing downloaders is to – unintentionally – financially support PPI downloaders, in that attackers are paid for each installation. However, the cash flow in this case is that attacker A (whose downloader is executed in Sandnet) is paid by attacker B (who asked for his malware being dropped), not causing harm to any innocent uninvolved individual.

*Next steps:* Until now, we mainly focused on characterizing downloaders, observing their infrastructures and analyzing downloaded eggs. Due to space and time constraints, we did not explore and include all analyses on the downloaded eggs, which we plan to work on as a follow-up of this paper. We also plan to extend our large downloader dataset for active techniques, e.g., to measure the prevalence of downloaders or to explore possible detection/mitigation techniques.

## 7 Conclusion

We identified and characterized 23 downloader families, showing that the downloader landscape is diverse in terms of architectural design, communication protocols and encryption schemes being used. We observed that many downloaders – albeit sometimes simple – have been actively operated for more than a year. Motivated by this observation, we analyze how attackers ensure the resilient operation of their downloader infrastructure. For example, we show that downloaders migrate their C&C servers aggressively among different Autonomous Systems, often involving multiple countries. Similarly, we observed downloaders not only to alter the C&C domains frequently, but also to involve diverse domain registrars. We revealed further details on the workings of downloaders, such as server-side polymorphism, by analyzing the download server infrastructure. These observations show that mitigating the problem of downloaders is more difficult than it might seem. To foster future research in this area, and as automated mechanism to acquire previously unseen malware samples, we present two generic techniques which extract downloaded eggs from any downloader.

## References

[1] Antivirus tracker, `http://avtracker.info/`
[2] yara-project - A malware identification and classification tool, `http://code.google.com/p/yara-project/`
[3] Lelli, A.: Zeusbot/Spyeye P2P Updated, Fortifying the Botnet, `http://www.symantec.com/connect/blogs/zeusbotspyeye-p2p-updated-fortifying-botnet`
[4] Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: 15th EICAR Conference (2006)
[5] Caballero, J., Grier, C., Kreibich, C., Paxson, V.: Measuring Pay-per-Install: The Commoditization of Malware Distribution. In: 20th USENIX Security Symposium, San Francisco, CA (August 2011)
[6] Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: 15th ACM Computer and Communications Security Conference, Alexandria, VA (October 2008)
[7] John, J.P., Moshchuk, A., Gribble, S.D., Krishnamurthy, A.: Studying Spamming Botnets Using Botlab. In: NSDI (2009)

[8] Kirat, D., Vigna, G., Kruegel, C.: BareBox: Efficient Malware Analysis on Bare-Metal. In: Proceedings of the Annual Computer Security Applications Conference, ACSAC (2011)
[9] Nazario, J., Holz, T.: As the Net Churns: Fast-Flux Botnet Observations Tracking Fast-Flux Domains. In: 3rd International Conference on Malicious and Unwanted Software, Malware 2008 (2008)
[10] Neugschwandtner, M., Milani Comparetti, P., Platzer, C.: Detecting Malware's Failover C&C Strategies with SQUEEZE. In: 27th Annual Computer Security Applications Conference, ACSAC, Orlando, Florida (December 2011)
[11] Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: Automatic Protocol Replay by Binary Analysis. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006 (2006)
[12] Rossow, C., Dietrich, C.J., Bos, H., Cavallaro, L., van Steen, M., Freiling, F.C., Pohlmann, N.: Sandnet: Network Traffic Analysis of Malicious Software. In: ACM EuroSys BADGERS (2011)
[13] Golovanov, S., Rusakov, V.: TDSS, http://www.securelist.com/en/analysis/204792131/TDSS
[14] Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., Vigna, G.: Your Botnet is My Botnet: Analysis of a Botnet Takeover. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009 (2009)
[15] ThreatExpert, http://www.threatexpert.com

# Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications

Steve Hanna[1], Ling Huang[2], Edward Wu[1], Saung Li[1],
Charles Chen[1], and Dawn Song[1]

[1] UC Berkeley
[2] Intel Labs

**Abstract.** Mobile application markets such as the Android Marketplace provide a centralized showcase of applications that end users can purchase or download for free onto their mobile phones. Despite the influx of applications to the markets, applications are cursorily reviewed by marketplace maintainers due to the vast number of submissions. User policing and reporting is the primary method to detect misbehaving applications. This reactive approach to application security, especially when programs can contain bugs, malware, or pirated (inauthentic) code, puts too much responsibility on the end users. In light of this, we propose Juxtapp, a scalable infrastructure for code similarity analysis among Android applications. Juxtapp provides a key solution to a number of problems in Android security, including determining if apps contain copies of buggy code, have significant code reuse that indicates piracy, or are instances of known malware. We evaluate our system using more than 58,000 Android applications and demonstrate that our system scales well and is effective. Our results show that Juxtapp is able to detect: 1) *463* applications with confirmed buggy code reuse that can lead to serious vulnerabilities in real-world apps, 2) *34* instances of known malware and variants (*13* distinct variants of the GoldDream malware), and 3) pirated variants of a popular paid game.

## 1 Introduction

As mobile devices (e.g., smartphones, tablets) gain popularity, software marketplaces have become centralized locations for users to download applications. For the Android operating system, Google hosts the official Android Market while Amazon and many others provide third party markets. The wide range of devices that are Android-compatible combined with the open source nature of the Android operating system and development platform have led to explosive growth of the Android market share. As of August of 2011, Android has grown to a 52% market share[33].

The rapidly increasing volume of applications, increased demand for diversified functionality, and existence of piracy and malware places large obstacles in the way of a healthy and sustainable Android market.

**Vulnerable Code Reuse.** Android developers often misuse coding idioms in Android, either due to copying and pasting of vulnerable code or lack of developer understanding [20,18]. For instance, Google has provided sample code to interface with the License Verification Library and the In-Application Billing APIs, which are responsible

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 62–81, 2013.

for verifying that a user is authorized to execute a program and purchasing virtual items within an application, respectively[7,6]. Google explicitly warns developers that they need to modify certain parts of the code, because the unmodified template code is subject to certain security vulnerabilities and requires developer intervention in order to ensure security properties.

**Malware.** With the exploding growth in the number of Android applications, the occurrence of Android malware has also increased. As of August 2011, users are 2.5 times more likely to encounter malware on their mobile devices than only 6 months ago and it is estimated that as high as 1 million users have been exposed to malware[11].

**Piracy.** Furthermore, the Android software marketplaces are home to many pirated applications. A common occurrence is for an illegitimate author to repackage and rebrand a paid or popular app with additional program functionality in order to generate revenue and even execute malicious code[4].

The current markets usually rely on two approaches to identify and remove potentially dangerous applications: 1) review-based approach, which requires mostly expert manual review and security examination, and 2) reactive approach, e.g., user policing, reporting, and user ratings as indicators that an application may be misleading in its functionality or misbehaving. Given the existence of hundreds of thousands of applications on the markets, neither approach is scalable and reliable enough to mitigate threats to users. To empower and expedite this process, we need an automated analysis of Android applications in order to pare down large application datasets into a small set of noteworthy candidates for further investigation.

Each of the aforementioned problems appears to be unrelated. However, we observe a common invariant among them, namely, code reuse, which sheds light on the fact that a unified approach in detecting common code (or code similarity) may address all of our goals. Using this observation, we propose to build a fast and scalable infrastructure for detecting code reuse in Android applications which allows for 1) early detection and developer notification of known vulnerable or buggy code, 2) detection of instances of known malware, either in isolation or repackaged with an innocuous program, and 3) detection of pirated applications.

It is a challenging task to develop a system to automatically detect code reuse in Android applications. The system must be able to quickly compare code and detect reuse, and scale to hundreds of thousands applications or more; the system need to be resilient to certain levels of code modification and obfuscation, which are common in Android applications; the system should be able to represent the application being compared in a meaningful, accurate way in order to find the so-called needle-in-a-haystack differences in applications, all the while maintaining low false positive and false negative rates.

As a first step solution, we use $k$-grams of opcode sequences of compiled applications and feature hashing[26,24] to efficiently tackle the problem at large-scale. $k$-grams of opcode sequences have been shown to be resilient to certain types of code modification and can be efficiently extracted from applications. Additionally, feature hashing has been shown to work well in dimensionality reduction and classification. We combine this technique with a variety of domain-specific knowledge in order evaluate code reuse, instances of known malware, and piracy in Android applications. We use $k$-grams and feature hashing combined in order to have a robust and efficient representation

of applications. Using this representation, we have a fast way to compute pairwise similarity between applications to detect code reuse among hundreds of thousand of applications.

In this paper, we propose Juxtapp, a scalable architecture for quickly detecting code reuse and similarity in Android applications. We implemented our distributed architecture using Hadoop and ran it on Amazon EC2. It is capable of fast, *incremental* additions to the analysis dataset, meaning it is amenable to frequent updates and additions to the pool of applications. We apply Juxtapp to address three different types of problems: vulnerable code reuse, known malware, piracy. We evaluate Juxtapp's ability to detect these problems on 58,000 applications, ranging in size from hundreds of kilobytes to tens of megabytes, which were collected from the official Android market and the Anzhi third party market[1]. We find that the system performs and scales well.

- *Vulnerable Code Reuse*. We show that applications widely use significant portions of the Google In-App Billing and License Verification example code, leaving them susceptible to vulnerabilities.
- *Instances of Known Malware*. We find **34** instances of malware in Android markets, **13** of which are distinct, previously unknown variants that have been repackaged with innocuous-looking applications.
- *Piracy*. We identify pirated applications in third party markets and show that Juxtapp can detect pirated applications that are obfuscated and with significant code variation from the original application.

## 2 Problem Definition

In this paper we consider the problem of automatically finding similarity among Android applications with the goal of detecting known buggy code patterns and vulnerabilities, repackaged and pirated applications, and known malware in Android markets. Detecting code reuse in Android applications offers a first chance in detecting applications that may negatively impact the user's security and experience or defraud developers of revenue. We develop Juxtapp, an architecture that automatically examines code containment in Android applications. We define code containment to be a measure of the relative amount of code in common between two Android applications. Using this, we examine a variety of Android market applications for instances of vulnerable code, known malware, and piracy.

**Buggy and Vulnerable Code Reuse**. Previous manual investigations into developer errors[18,20] in Android applications have indicated that developers often copy and paste code as well as reuse sample code obtained from Android-specific developer websites without modification. Using application similarity, we can examine the Android Market to see if they contain known buggy or vulnerable pieces of code.

**Known Malware**. The incidence of malware in Android marketplaces has been rising rapidly. In January 2011, 80 applications were known to be infected with malware, as opposed to June 2011, when the incidence rate had risen to over 400 instances of malicious applications [11]. Malware authors often repackage legitimate applications with a malicious payload in order to entice users to download an infected application.

**Piracy and Application Repackaging**. Popular Android applications and games are commonly repackaged with modified code in order to evade copyrights protection and to generate revenue for the pirate [4]. By comparing applications from the official Android market to third party markets we show that we can detect instances of piracy.

**Scope**. We restrict ourselves to the Android application domain, excluding obfuscation in the form of functional code transformation. For instance, we are able to detect two instances of similar obfuscated code, but we restrict ourselves to this domain and do not consider the problem of matching code which has been transformed to be functionally equivalent.

### 2.1 Goals and Challenges

Juxtapp has a variety of challenges which must be met in order to detect code reuse in Android applications. Some specific goals of our platform are to:

**Automatically Analyze Code Similarity in Android Applications.** As of November 2011, the Android market had over 310,000 applications[12]. The rapid growth of market applications and increase in the number of pirated and malicious applications underlines the need for a way to rapidly and automatically analyze applications.

**Scale to a Large Number of Applications.** Android markets have hundreds of thousands of applications with new applications being added all the time. Our architecture must be able to scale in order to detect similarity across a wide range of applications, including the ability to*incrementally* update our application repository in an efficient manner.

**Accurately and Efficiently Represent the Applications under Analysis.** In handling hundreds of thousands of applications, Juxtapp must be able to accurately represent and quickly determine code similarity among applications. There is an implicit tradeoff between the accuracy of the analysis and the amount of space it takes to represent an application under analysis.

**Android Specific.** In addition to general challenges, there are a number of domain specific considerations when computing the similarity of Android applications.

**Java Source Code Unavailable.** For most applications on the Android Markets, source code is not available. Android applications are compiled from Java to Dalvik bytecode (known as the DEX format)—the bytecode for the Dalvik VM[3]. This compiled code and application resources are packaged as an APK. The DEX format fully describes the application and retains class structure, function information, etc.

**Multiple Entry Points.** Unlike traditional desktop programs, Android applications have multiple potential entry points. Android applications are broken up into components and these components can each have their own entry points.

**Obfuscation.** Android developers are encouraged to obfuscate their code using Proguard [13]. Proguard attempts to remove unused code and renames classes, methods, and fields with obscured names to make reverse engineering of Android applications more difficult. However, this process is deterministic so two identical applications will be transformed in the same way.

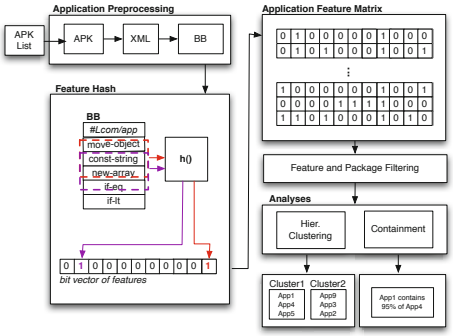



**Fig. 1.** The Juxtapp Workflow

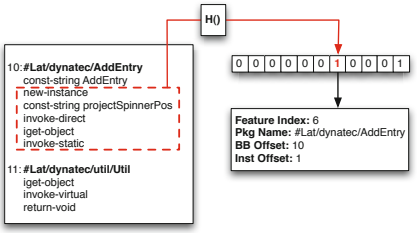



**Fig. 2.** Example outcome of feature hashing a basic block found in an application

Therefore, any type of program analysis must take these challenges into consideration. And indeed, the domain specific challenges can be used to impose structure on the applications so that feature hashing and clustering are more amenable in the Android application domain.

## 3 Background

Like static code reuse detection proposed in [32,28], we use $k$-gram features of code sequence to represent applications. However, $k$-grams extracted from code sequence usually results in an enormous feature space (e.g., $2^{128}$ features in [32,28]), preventing efficient feature storage and similarity comparison even for a moderate number of applications. To analyze large volumes of mobile applications, we need an efficient feature representation of the applications and a fast way to compare features between them.

**Feature Hashing.** The main technique we use is feature hashing. Feature hashing is a popular and powerful technique for reducing the dimensionality of the data being analyzed [26,31]. Using a single hash function, it compresses the original large data space into a smaller, randomized feature space, in which feature hashing, representation, and pairwise comparison are all efficient. This efficiency comes at the cost of potential collisions while hashing. However, theoretical and experimental results from the machine learning community show that pairwise similarity maintains high accuracy, thus algorithms built on top like hierarchical clustering, will be close to exact [26,31]. Feature hashing was recently introduced into the security community for malware analysis [24].

The resulting representation of an application can be encoded into a succinct bitvector which represents the features present in the data. As always, choosing a good hash function and a bitvector representation of prime length is essential to minimize the number of collisions in the vector.

**Similarity.** We determine the similarity of two applications by the similarity between their feature sets. We use the Jaccard similarity metric defined as $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where $A$ and $B$ are two $k$-gram feature sets of two applications, respectively. Because

we hash $k$-gram sets into boolean vectors, with each entry indicating presence or absence of a feature, as opposed to a set of items, we can approximate this quantity much more efficiently using bit-wise operations: $J(\hat{A}, \hat{B}) = \frac{|\hat{A} \wedge \hat{B}|}{|\hat{A} \vee \hat{B}|}$, where $\hat{A}$ and $\hat{B}$ are bitvector representations of $k$-gram sets $A$ and $B$, respectively. As shown in [24], as long as the size of the bitvector is large enough, $J(\hat{A}, \hat{B})$ is very close to J(A,B), the similarity between two applications represented by the $k$-gram feature sets. The Jaccard distance $D(A, B)$, which measures *dissimilarity* between two feature vectors, is obtained by subtracting the Jaccard similarity from one: $D(\hat{A}, \hat{B}) = 1 - J(\hat{A}, \hat{B})$. Both Jaccard similarity and distance have values in the range $[0, 1]$.

## 4 Our Approach

As shown in Figure 1, our approach, Juxtapp, involves the following steps for analyzing Android applications: 1) application preprocessing, 2) feature extraction, and 3) clustering and containment analyses.

### 4.1 Application Preprocessing

We preprocess each application in order to extract the DEX file, which represents the compiled application code. In our approach, the original Java source code is *not required* because the DEX format fully describes the application and retains class structure, function information, etc.

For each application we convert its DEX file into a complete XML representation of the Dalvik program, including program structure. From this, we extract each basic block and label it according to which package it came from within the application. We process each basic block and only retain the opcodes while discarding most operands. The exception to this is for opcodes storing constant data, such as the `const-string` opcode, which becomes a concatenation of the opcode along with the value it references.

The intuition behind this is that many Java applications contain boiler plate code that will appear in many applications when only opcodes are considered. Furthermore, including constants makes the feature hashing (discussed below) more fine-grained and more restrictive about matching. This is especially important because many applications use Java reflections to access functionality, with the only difference between programs being the string arguments passed to the Reflections API.

### 4.2 Feature Extraction

We use $k$-grams of opcodes and feature hashing to extract features from applications. We use the `djb2` hash function which is known to have an excellent distribution[9]. As shown in the *Feature Hash* box in Figure 1, for each application's basic block representation of the original XML file, file), we extract each $k$-gram using a moving window of size $k$, and hash it using `djb2`. $k$-grams across basic blocks are ignored. For each hashed value, we set the corresponding bit in the bitvector of the application, indicating existence of the $k$-gram. Along with this information, we efficiently store the package name from which the basic block originated, the basic block offset within the basic

block file, and the $k$-gram offset. This allows us to recover how and why our architecture determined that applications are similar and serves as a way to verify matching applications.

In order to feature hash, we have two parameters to determine, namely: length of $k$-gram $k$ and bitvector size $m$. In Section 5, we show an experimental evaluation of several values of $k$ and $m$ in order to determine optimal values of these parameters over our dataset.

**Choosing $k$.** $k$ is a parameter which determines the number of dimensions of the underlying feature space for representing the Android applications, and it bounds the number of features that can be extracted for each application. $k$ is a crucial parameter for detecting similarity. If $k$ is too small (e.g., $k = 1$), there will be a small number of unique features from all applications, resulting in an oversimplified, low-dimensional representation of the applications. In this representation, overmatching between applications can occur, and many applications would be falsely classified as being similar applications would be falsely similar. On the other hand, if $k$ is too large (e.g., bigger than the size of most basic blocks), even small code changes might result in large changes in the feature representation, preventing us from obtaining a meaningful and robust comparisons between applications. In general, a reasonable $k$ should have a small value, at which further increase in value would cause insignificant increase in the quality of the pairwise similarity comparison. As shown in Section 5, we evaluate different $k$ values, and choose $k = 5$, at which its marginal impact on similarity accuracy is around 0.01.

**Choosing Bitvector Size.** The bitvector size $m$ strikes the tradeoff between (similarity) computation efficiency and approximation error of the bitvector representation of the $k$-gram features.

Ideally, we want size $m$ to be large enough so that few collisions would happen when we feature hash $k$-grams into bitvectors; practically, we want size $m$ to be small so that we can efficiently compute pairwise similarity among hundreds of thousands of applications. The larger the bitvector size $m$, the more accurately a bitvector represents an application, but at the cost of more time required to compute the pairwise similarity among all applications.

As shown in [24], as long as $m \gg N$, which is the number of $k$-grams extracted from an application, the Jaccard similarity between two bitvectors very closely approximates computing the set intersection between two $k$-gram feature sets. That is, as long as $m$ is large enough, Jaccard similarity is nearly an exact representation in practice. Based on this principle, we use a data-driven approach in our experiments in Section 5, in order to determine a bitvector size which is large enough to represent the feature space in question.

### 4.3 Analysis of Feature Hashing Results

A variety of data analyses can be performed on the feature representation of the applications. In this paper, we primarily focus on similarity, containment and clustering analysis, which help us to filter out vast amounts of uninteresting instances and pare down a small set of interesting candidates for further analysis.

**Code Containment Comparison.** Containment analysis is a useful tool for paring down application candidates that potentially have copied code, pirating, and malware contamination. We define the containee $A$ to be the application being examined for similarity and the container (or carrier [1] ) $B$ to be the application which houses the packages and associated features that we test for existence inside the containee. We define a metric that gives the percentage of containment by considering the number of features common in both applications, divided by the number of bits in the containee application. Formally, containment is defined as: $C(\hat{A}|\hat{B}) = \frac{|\hat{A} \wedge \hat{B}|}{|\hat{A}|}$. Written in this form, this containment is defined as the percentage of features in application $A$ that exist within application $B$.

**Clustering.** To find inherent patterns among Android applications, we use agglomerative hierarchical clustering[19] on the feature bitvector representation of each application in order to group similar applications together. The basic idea is that the collection of feature bitvectors represents the applications in a high-dimensional space with a well-defined distance metric, the Jaccard distance. Using this distance metric, we can group bitvectors that are close-by and, thus, we are able to group similar applications.

Hierarchical clustering produces clusters without having to specify the number of clusters in advance. The input to the clustering algorithm is a threshold $t$ (e.g., 90%) and a list of Jaccard similarity values between each pair of applications. The output is a clustering $S$ for the applications, in which all applications in a cluster are with similarity $s$ greater than or equal to $t : s \geq t$. The threshold $t$ is set by the desired precision tradeoff between the number of applications in the clusters and the "closeness" of applications within a given cluster. While a smaller $t$ puts more applications into a few large clusters, a larger t discovers specific variants of application families (e.g., similar applications developed by the same authors).

Hierarchical clustering begins with one application in its own cluster; then it selects the closest pair and merges them into a common cluster. The cluster comparing and merging process continues until there is no pair whose similarity exceeds the input threshold $t$.

### 4.4 Core Functionality and Result Refinement

Clustering can be a way to visualize the application topology in order to qualitatively understand how well applications are classified among a given cluster. Application similarity can be dominated by large similar libraries common to many applications (i.e. AdMob). In light of this, we develop the notion of *core functionality*, which seeks to capture in a coarse-grained manner how included libraries interact with the main application component.

Simply put, we examine each application and whether or not the core application component directly invokes an outside library. If it does then it is a part of the application's functionality; otherwise, that code can be excluded from our analysis. We refer to the set of libraries excluded as an *exclusion list*. We point out that this is an

[1] In the case of malware, a carrier is a more appropriate term because the innocuous application is modified in order to execute code outside of the intended functionality.

over-approximation and aggressively excludes libraries due to Java reflection as well as dynamically registered event handlers, and other entry points defined by the Android Manifest.

### 4.5 Implementation

The workflow of Juxtapp can be roughly broken up into the following stages: application preprocessing, feature hashing, clustering and containment analysis. Juxtapp consists of 6,400 lines of C++, 1,600 lines of Java, and 600 lines of scripts.

The first step in the process is converting the Android application file (APK) to a format which is usable by our architecture. Juxtapp processes the APK and outputs the file in an XML format with functions split into basic blocks, which is then converted to a basic block format, which has a label indicating the source package, class and method.

After preprocessing, the applications are feature hashed. Juxtapp processes the basic block file for each application and outputs a feature vector representing the application along with recovery information to verify matching portions of applications. That is, in addition to the features, we also store the package and class name, and the offset within the original file in order to verify matching potions of applications. Figure 2 shows an example basic block being feature hashed, along with the recovery information we store for each feature. For each program under analysis, the features calculated are stored as a sparse representation of the vector, while a table of each feature's offset within the original program along with the package and class from which it originated.

After processing all of the applications' basic block files, Juxtapp calculates a pairwise distance matrix between all applications. This matrix is used for clustering and determining similarity among applications.

After the applications have been feature hashed, Juxtapp can perform other in-depth analysis. First, the applications under analysis can be clustered based on their computed distance matrix, which offers a topological view of the dataset, which can help an analyst narrow down interesting areas to investigate.

Finally, Juxtapp computes containment between sets of applications. Given a set of feature hashed applications represented, the containment tool determines what features are common between applications and outputs the percentage of code in common, along with the ratio of the comparative sizes of the number of features. The intuition behind this is that a large application when compared to a small application may inadvertently have a large subset of the smaller applications features by virtue of the fact that a larger application will produce a dense feature vector. This ratio is used to remove false positives.

**Distributed Analysis.** We have both a single machine implementation of Juxtapp as well as a distributed implementation which uses Hadoop on Amazon EC2. We use the Hadoop MapReduce framework for performing large-scale computations and HDFS for sharing common data among nodes[8]. We wrote a MapReduce application in order to perform the APK to Basic-Block conversion portion of the workflow, and we used Hadoop Streaming to interface with our C++ applications, which were responsible for feature hashing and containment calculations. Many of the tasks required of Juxtapp are easily parellizable tasks which greatly improves performance when dealing with large

datasets. As a result, Juxtapp can feature hash, cluster, and analyze containment in a distributed manner which offers great performance increases over the single machine version.

**Incremental Update and Increasing Performance.** The statelessness property of many stages in Juxtapp makes it easy to incrementally process the applications, update their similarity matrix, and analyze them in detail without the need to reprocess all applications under analysis. When creating or updating the pairwise similarity matrix, only values greater than *50%* similarity are stored, making the matrix sparse among dissimilar applications. When a set of $m$ new applications are added to the analysis, the application preprocessing (conversion of APK to XML to Basic Block) and the feature hashing are inherently incremental, meaning, only the new applications need conversion and feature hashing. As shown in Figure 3, with $n$ existing applications and $m$ new applications, updating the existing $n \times n$ similarity matrix $A$ is straight forward as follows: 1) compute the $m \times m$ similarity matrix $B$ among the new applications, 2) compute the $n \times m$ similarity matrix $C$ between the set of new applications and the existing ones, and 3) concatenate them together and grow the existing similarity matrix $A$ at appropriate rows and columns to get the new $(n+m) \times (n+m)$ similarity matrix.

## 5 Evaluation

In this section, we evaluate the efficacy of Juxtapp. We first introduce our evaluation dataset and describe our experimental setup. Then, we discuss determining experimental parameters and their impact on our results. Finally, we introduce case studies in which we use Juxtappto detect instances of vulnerable code reuse, known malware, and piracy on Android markets.

### 5.1 Experimental Evaluation Dataset

We evaluate our approach using applications from three different sources. From the official Android Market we obtained 30,000 free Android applications. Additionally, we downloaded 28,159 applications from a third-party Chinese market, Anzhi [1], and the 72 malware in our malware dataset came from the Contagio malware dump and other sources [2]. Lastly, we use a set of 95,000 Android applications from the official Android Market to evaluate the performance of Juxtapp [2].

### 5.2 Experimental Setup and Performance

Local experiments, when tractable, such as containment between a small set of applications and our dataset, were run on Ubuntu Linux 2.6.38 with Intel Xeon CPU (8 cores) and 8GB of RAM. When larger experiments were required, such as containment between on-market to off-market applications, and generating pairwise distance matrix, we conducted them on Amazon EC2. For our Amazon EC2 clusters, we used

[2] We obtained a larger dataset of applications in order to show that our technique scales to a large number of applications beyond our evaluation set of applications

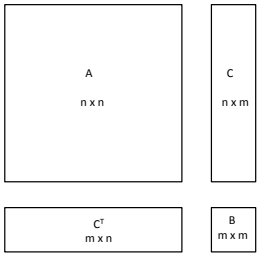



**Fig. 3.** Incrementally updating the similarity matrix. $A$ contains similarity values among existing applications, $B$ contains similarity values among the new applications, and $C$ contains similarity values between the new applications and the existing ones.

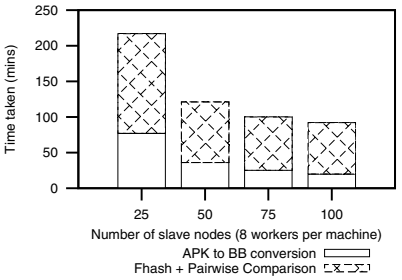



**Fig. 4.** The running time of our complete pipeline with various number of workers per cluster on Amazon EC2. Time are measured when processing 95,000 unique Android applications.

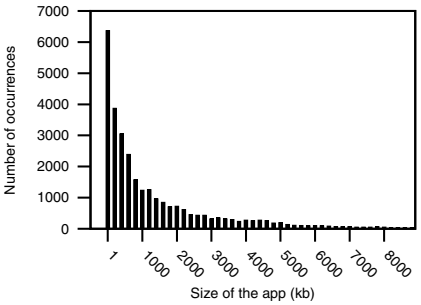



**Fig. 5.** Frequency distribution of size of APK files. For better visualization, we do not show the largest 4% of the applications. The largest APK file in our dataset is 52.26MB. The numbers on the x-axis are the lower bounds of the bins, and the size of each bin is 200.

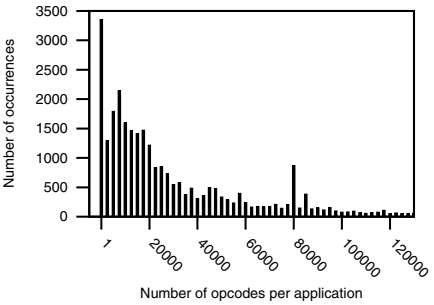



**Fig. 6.** Frequency distribution of number of opcodes of applications. For better visualization, we do not show the largest 11.2% of basic block files. The largest file in our dataset has 1,728,196 opcodes. The numbers on the x-axis are the lower bounds of the bins, and the size of each bin is 2500.

m2.4xlarge instances, which run on Ubuntu Linux 2.6.38-8-virtual with 8 virtual cores and 68.4GB of memory.

We varied the number of nodes running from 25 to 100 and used 8 worker threads per node. Figure 4 shows the time required to complete a full run of the entire pipeline, which includes APK to basic block format file conversion, feature hashing, and computation of the pairwise similarity when using 95,000 unique Android applications. At the time of writing, there are around 310,000 Android Applications[12], which demonstrates that Juxtapp scales well.

As we increase the number of nodes, the amount of time required to do analysis becomes gradually dominated by the overhead of parallelization. In addition, the APK to Basic-Block and feature hashing stages were parallelizable without any synchronized

state, which contributed to significant performance gains as the number of workers increased. However, the pairwise distance comparison is the current bottleneck on performance because it combines the resulting bitvectors from each worker. Figure 4 shows how the overhead of the pairwise comparison approaches a constant overhead as the number of nodes are increased.

**Incremental Update Performance.** Incremental updates of the dataset allow us to continuously process and update our dataset with new market applications without requiring running the entire Juxtapp workflow on our application repository. Table 1 shows the time required to add from 100 to 7,000 APKs to the dataset. Distribution time is the time required to distribute APKs to worker nodes. This time begins to become dominant as the number of APKs increases. This overhead is caused by not being fully able to take advantage of Hadoop's resource allocation, due to our Hadoop Streaming implementation. Despite this, these numbers show that adding a large number of applications to the comparison repository daily or even multiple times daily is feasible with Juxtapp.

**Table 1.** The time to incrementally process varying numbers of APKs. Note, distribution time is included to show how file distribution starts to dominate the processing time.

| # Incr. APKs | Distribution Time | Completion Time |
|---|---|---|
| 100 | 0m 36s | 5m 11s |
| 500 | 4m 49s | 9m 35s |
| 1000 | 8m 58s | 21m 5s |
| 3000 | 20m 20s | 42m 31s |
| 5000 | 42m 52s | 80m 51s |
| 7000 | 57m 0s | 104m 48s |

**Table 2.** Experiment showing the impact of varying $k$ on the Jaccard distance

| $k$ | Avg. Dist |
|---|---|
| 3 | 0.939 |
| 5 | 0.969 |
| 7 | 0.980 |
| 9 | 0.984 |

### 5.3 Dataset Statistics

To gain a general understanding of our dataset, we analyzed our collection of 30,000 unique applications as a representative sample of the official Android Market. Figure 5 shows the distribution of the sizes of APK files in kilobytes, and Figure 6 shows the distribution of the number of opcodes per application. Both distributions are skewed to the right, with APK files having a median size of 724KB and applications having a median number of opcodes of 20,555. The 75th percentile values for APK file sizes and number of opcodes are 2,071kb and 56,166, respectively. The total file size of these APKs is 50.43GB and total number of opcodes in all applications is approximately 1.45 billion.

### 5.4 Determining Experimental Values

Before feature hashing we must choose values for $k$-gram size $k$ and bitvector size $m$. We use the 30,000 Android applications to determine their values.

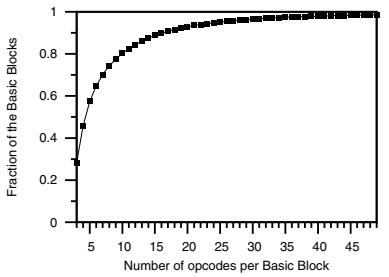



**Fig. 7.** Cumulative distribution of the number of opcodes per basic block, using all basic blocks with more than 2 opcodes. The mean is 5.35 opcodes and the median is 2 opcodes, while the largest one contains 35,517 opcodes.

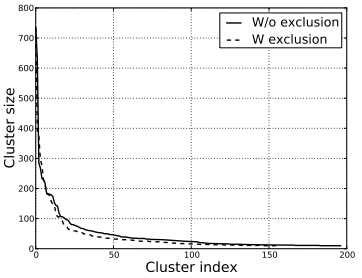



**Fig. 8.** The clusters obtained from 30,000 Android application using hierarchical clustering with similarity threshold $t = 0.9$

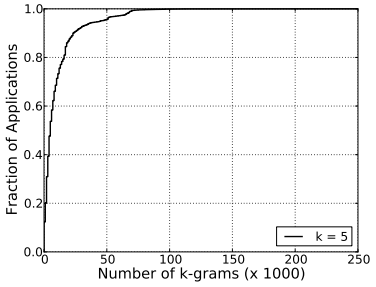



**Fig. 9.** Cumulative distribution of the number of unique $k$-grams extracted from 30,000 Android applications (with $k = 5$)

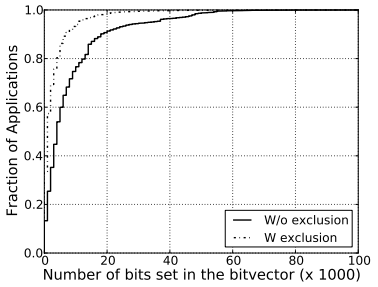



**Fig. 10.** Cumulative distribution of the number of bits set in the bitvectors of 30,000 Android applications

**Choosing $k$ for Our Dataset.** To choose $k$, we randomly select pairs of applications and evaluate their Jaccard distance to determine how much varying $k$ impacts the average distance between them. Figure 2 shows varying values of $k$ and the resulting average distance between pairs of randomly sampled 6,000 applications[3]. We repeat the experiment on multiple runs, but see little variance across them. The key intuition is if two applications are chosen at random from our dataset, they are likely to be dissimilar. The table shows that starting from 5, further increasing $k$ has little impact on the distance calculation. Based on this, we chose a value of $k$ to be 5 and performed feature hashing and clustering on our sampled applications. Figure 7 shows the cumulative distribution of opcodes per basic block for all basic blocks with more than two opcodes. This indicates that the majority of the basic blocks are dominated by a small value of $k$, and *5* is an appropriate choice for this dataset.

[3] A distance of 1 indicates no similarity where a distance of 0 indicates identical similarity.

**Choosing an Appropriate Bitvector Size.** The bitvector size $m$ strikes the trade-off between efficiency (similarity) computation and approximation error of using bitvectors to represent the sets of $k$-gram for each application. Ideally, we want size $m$ to be large enough such that few collision occur during feature hashing. Practically, we want size $m$ to be small so that we can efficiently compute similarity between all pairs of hundreds of thousands of applications.

According to [24], we need $m \gg N$, the number of $k$-grams extracted from an application, so that the Jaccard similarity between two bitvectors is very close to the exact representation of computing the intersection between two $k$-gram feature sets. In addition, in all of our analysis, we are particularly interested in applications with high similarity, e.g., *application pairs with similarity greater than 50%*. We sparsely store this pairwise matrix and only store values for which the threshold is reached. This optimization yields good similarity results because those excluded applications are very unlikely to have a similarity score greater the threshold with other apps.

We use all of the 30,000 applications from the Android Market to determine $m$. We compute the number of unique 5-gram features that can be extracted from each application, and plot its cumulative distribution from all applications in Figure 9. We find $N_{90}$ in the distribution, which represents the threshold in which 90% of all applications' $k$-gram features are less than this value. We then set $m = 240,007$, a prime that is more than nine times of $N_{90}$, satisfying the condition $m \gg N$ suggested by [24].

We use the following two ways to verify whether $m = 240,007$ is large enough. 1) We do feature hashing with $m = 240,007$ for all 30,000 applications, and count the number of bits set in the bitvector for each application. We plot its distribution in Figure 10. We observe that more than 95% of applications have $1/5$ of their bits set in their bitvectors, and more than 90% of applications have only $1/10$ of their bits set. Hence, we do obtain sparse bitvector representation for the majority of applications. 2) We also randomly sample a subset of 1000 applications, compute the pairwise similarity among them using their $k$-gram feature sets, and compare the similarity values to those computed using their bitvectors. We find that the average difference between them is less than 0.01. With these two observations, we conclude that $m = 240,007$ is suitable for our analysis.

**Clustering for Application Topology.** We use clustering as a way to group similar applications together. We run hierarchical clustering on the 30,000 Android applications using a similarity threshold $t = 0.9$, with and without a core functionality exclusion list applied, respectively. Figure 8 shows the cluster size sorted in a descending order. We see that both versions of the clustering worked well, but clusters with exclusion no longer had application clusters dominated by large libraries.

We observe that there are around 200 clusters, each of which has at least 10 applications, and in total there are 9344 applications in those clusters. We find that our clustering identified three unique, commonly occurring patterns. They are:

**Same Application Title, Different Versions.** One cluster contained several versions of the same movie player, which were all responsible for displaying elicit pictures of models. Within the cluster, there were *4* different versions of the same model's movie player, heaven8.

**Differing Author and Functionality, Same Tool for Development.** In one example, AppBar is a tool for allowing users to visually create applications for Android without needing to know about the underlying development platform. The platform allows for the addition of sounds, images, twitter feeds, and all sorts of additional widgets. We identified a cluster on the official Android market consisting of *735* applications of this type, ranging from RSS feeds to audio programs.

**Multiple Apps from an Author, Different Underlying Functionality.** A common pattern is for a developer to make a framework for creating applications and then reusing the applications in a variety of contexts. For instance, the company BrightAI produces a variety of applications related to sports. One such cluster contained *28* different applications, all by BrightAI, but with different application purposes.

### 5.5 Case Studies

Previous work on studying Android applications[20] has shown that developers copy and paste code snippets from popular programming web sites into their own code, without understanding the potential security risks posed by blindly copying code.

Recently, Google announced an In-Application Billing API along with a sample application which demonstrates how the purchasing protocol works[10]. Several security warnings accompany the document, including statements regarding how developers should obfuscate their code, protect their purchasable content, and verify purchases on a remote server. We show how Juxtapp can not only detect applications in the Android Market that copied this sample code, but we also show how we can detect other known source code-related vulnerabilities in the market using our architecture.

**Reuse of Vulnerable Code.** In this section, we examine two cases of vulnerable code reuse of sample code provided by Google: In-Application Billing and the License Verification Library. We show that Juxtapp can quickly and efficiently reduce the set of potentially vulnerable applications and detect vulnerable code reuse in Android Applications.

**In-Application Billing.** Google In-Application Billing (IAB) is a library provided for developers to include so that their customers can sell digital content within their application, while letting Google handle authentication and credit card purchases[6]. For security reasons, Google advises that developers use obfuscation in order to make the code more difficult to understand for an adversary and they also recommend that developers perform verification on a remote server.

However, the sample code provided by Google is not obfuscated and performs verification of a purchase on the device. The left side of Figure 11, Line 231, shows the potential single point of attack. Meaning, if a developer can rewrite the statement to negate the condition, or force it to be true in some other way, the application will skip verification and allow the current user access.

In order to detect a potential attack, we analyzed the containment between the IAB sample code and the 30,000 applications in our dataset. We set a threshold that at least 70% of the IAB sample code must be in the application before further exploration.

Running containment between the sample IAB code and the Android Market applications took *1.5 minutes*,and we detected **295** applications containing 70% of the IAB code. Other researchers used these applications to demonstrate that they could use the tool they developed for application rewriting to automatically exploit a vulnerability to get virtual goods for free [16]. Of those that used a significant portion of the sample code, **174** were vulnerable, while 65 use off-device/JNI verification and 56 were inoperable after rewriting. Our results show that Juxtapp is a fast way to quickly analyze large sets of applications for vulnerabilities caused by code reuse.

**License Verification Library.** The License Verification Library(LVL) is a library provided by Google in order to allow developers to query the Android Market at runtime in order to determine if a user is licensed to use a particular application[7]. Similar to IAB, Google provides sample code which encourages developers to obscure their code and ensure that single points of attack are protected. The sample code uses caching in order to prevent having to contact the Android Market every time the user invokes the application. However, the right side of Figure 11, Line 133, shows the potential vulnerability. This line could be rewritten to negate the condition, or to check another condition, making this a single point of failure, allowing a clever attacker to use the library without a license.

We executed containment on 30,000 using the Google LVL sample code to guide the search. For this experiment, we detected 272 potential candidates, **182** of which had 90% of the code, and **90** more, with at least 70% of the sample code. It took about *2 minutes* to analyze the dataset. Of the potentially vulnerable candidates, **239** of the 272 applications had the vulnerable pattern in their code. We manually verified the results in order to be assured that the pattern was in the code. Our analysis took about 10 minutes with script assistance responsible for opening each document which allowing the analyst to determine if the pattern exists, without the task of manually opening each file. Of those detected, some had obfuscated class and method names, but Juxtapp was still able to detect similarity.

```
222: boolean verify(...) {
231: if (!sig.verify(
232:         Base64.decode(signature))){
233:     return false;
234: }
235: return true;
```

```
130: void checkAccess(...) {
131: // skip asking market if cached license
133: if (mPolicy.allowAccess()) {
135:     callback.allow();
136: } else {
137:     //verification code
```

**Fig. 11.** The code on the top shows the vulnerable code present in the In-Application Billing Example Code `Security.java`. On the bottom is the point of vulnerability within the License Verification Library sample code `LicenseChecker.cpp`

**Android Malware.** The Android Market place has recently experienced an influx of malware. Google has responded by exercising its remote application removal ability, that is, if Google determines an application is malicious or untrustworthy, it can remotely push a command to remove the application from affected devices[5]. In fact, as of August 2011, users are 2.5 times more likely to encounter malware on their

**Table 3.** Number of instances of each kind of malware found in the Anzhi Market dataset. Also shown are the distinct new carriers discovered in our dataset.

| Malware | Instances Found | Distinct New Carriers Found | Malware BB Size |
|---|---|---|---|
| GoldDream | 25 | 13 | 1,898 |
| DroidKungFu | 6 | 0 | 5,357 |
| DroidKungFu2 | 2 | 0 | 375 |
| zsone | 1 | 0 | 280 |
| DroidDream | 0 | 0 | 2,526 |
| **Total** | **34** | **13** | - |

mobile devices than only 6 months ago, and it is estimated that as high as 1 million users have been exposed to mobile malware[11][14]. We suspect that unregulated, 3rd party markets will have a higher incidence of malware.

**Containment between Anzhi Market and Malware**. In order to evaluate whether third party markets contain known malware, we select a subset of 5 malware from our dataset, which represents some of the most prolific, well-known malware. They include: DroidDream, DroidKungFu1/2, zsone and GoldDream. Each malware sample had a manual exclusion list applied, that is, using widely available malware analysis, we excluded common code from malware such as advertising libraries and common utilities which contribute nothing to the uniqueness of the code

Table 3 shows that we were able to detect 34 malware in the off-market dataset. The experiment took around 10 minutes to complete. Among those that matched we noticed a very high incidence of code reuse ranging from 93%-100%. The lower percentage matching shows that the technique is amenable to code mutations and variants. When investigating those with lower percentages, we noted that variants often changed file paths, reworked small amounts of code, changed exploit names, etc., and a 100% match indicated, with high probability, that the two pieces of malware are identical and indeed, when investigated the samples matched.

When evaluating the samples we also consider the ratio of the malware sample compared to the container application. A low ratio indicates similar orders of magnitude among the code sample, where a higher ratio indicates that the reported matching is likely a false positive due to the density of the bitvector representing the larger application. Some malware found in the Anzhi market matched our sample malware dataset with little variation in code between them. However, other matched malware was significantly different from our evaluation set and we show how we can detect new variants, with new malware carriers using Juxtapp. Most of the minor changes were related to class and package names. However, Table 3 shows that we found 13 unique carriers of the GoldDream malware in our dataset. Meaning, of these we found **13** previously unknown to us, distinct applications in our evaluation dataset, which were all different types of games that had been repackaged with the GoldDream malware.

**Containment between Android Market and Malware.**We evaluated containment between 63 malware samples to the 30,000 collected from the official Android Market. The experiment took 19 minutes to execute locally.

Juxtapp did not detect any instances of known malware on the Android Market. This result is unsurprising given that Google has been vigilant about removing malware once it is found, banning the associated account, and issuing remote removal[15].

However, as expected, Juxtapp was able to detect the original application that the malware sample had been repackaged with in order to trick users into downloading. That is, a subset of our samples were repackaged with legitimate applications. Table 4 shows the Android applications we were able to detect using the malware sample.

**Table 4.** Juxtapp is able to detect the original (and versions) of the application which was repackaged when compared to our malware dataset. Multiple features indicate multiple versions in our dataset.

| Application File Name | Features | Name | Repackaged with |
|---|---|---|---|
| com.codingcaveman.solotrial.apk | 4,272/4,831 | Guitar Solo Lite | DroidDream.1 |
| it.medieval.blueftp.apk | 19,597/18,946 | Bluetooth File Transfer | DroidDream.2 |
| com.tencent.qq.apk | 28,712 | Tencent QQ Messaging | PJApps |
| de.schaeuffelhut.android.openvpn.apk | 2,009 | OpenVPN Settings | DroidKungFu |

**Piracy and Application Repackaging.** In addition to vulnerable code and malware on the Android markets, piracy, especially among games, has become a major problem for developers. Android applications are often pirated by rogue authors, which remove copy protection and replace developer revenue mechanisms such as advertising libraries. In order to examine the third party market Anzhi for piracy, we downloaded and paid for the two applications mentioned in the Guardian article about android privacy[4]: 1) Chillingo's The Wars; 2) Neolithic Software's Sinister Planet. We compared these applications against the 28,159 applications in the Anzhi market, which took around 19 minutes to execute locally.

We found no instances of the Sinister Planet program being pirated on a third party market. However, we found **3** pirated versions of Chillingo's The Wars, being marketed by the company Joy World, the same company accused of piracy in the article. Each of the pirated versions has **71%** code in common with the original application.

Despite the fact that the legitimate Wars program is unobfuscated, the Joy World version is obfuscated with methods and classes renamed. Additionally, we found that the pirate had added advertising libraries to the application which were not present in the original version. So, even in light of significant obfuscation and additional code added, we were still able to detect similarity showing that Juxtapp handles perturbations in code well.

## 6 Related Work

A technique similar to ours has been independently developed by Zhou *et al.*[34]. While they focused on detecting repackaged applications, we applied our technique and show that it is effective to detect repackaged applications, buggy code reuse and known malware. In addition, we implemented the technique on a distributed infrastructure using Amazon MapReduce, which enables us to analyze a much larger application corpus.

For large-scale malware analysis, Jang *et al.*[24] developed *BitShred*, a system for large-scale malware triage and similarity detection based on feature hashing. However, they focus on the technique as a contribution and classify x86 malware, whereas we apply similar techniques, with domain specific knowledge in order to find a variety of code reuse in Android marketplaces. Instead of using boolean features, Gao *et al.*[22] and

Hu *et al.*[23] use features based upon isomorphisms between control flow and function call graphs of the program. While these work primarily focus on techniques to compare and index malware, our work is focused on techniques to determine similarity among Android applications and conduct deep security analysis.

Winnowing, a fuzzing hashing technique that selects a subset of features from a program for analysis, has been widely used for code similarity analysis[17] and plagiarism detection[30]. However, the winnowing algorithm requires calculating set inclusion, which is expensive when comparing many features.

A variety of approaches for static code clone detection have been proposed in the programming language literature for refactoring, finding bugs, and better understanding of the code [21,27,29,25]. All those techniques can be applied into our framework to further improve the accuracy and robustness our approach.

## 7 Conclusion

In this paper we presented Juxtapp, a scalable architecture for detecting code reuse in Android applications. Our architecture is implemented in Hadoop and we ran it on Amazon EC2. We evaluated the efficacy of Juxtapp in detecting vulnerable code reuse, known malware, and piracy in a dataset of 58,000 applications from Android marketplaces. Our findings show that Juxtapp is a valuable architecture in detecting application similarity and code reuse in Android applications.

**Acknowledgments.** We would like to thank Adrienne Felt and Chris Grier for their insights with this paper.

## References

1. Anzhi android market, `http://www.anzhi.com/`
2. Contagio malware dump, `http://contagiodump.blogspot.com/`
3. Dalvik virtual machine, `http://www.dalvikvm.com/`
4. Developers express concern over pirated games on android market, `http://www.guardian.co.uk/technology/blog/2011/mar/17/android-market-pirated-games-concerns/`
5. Exercising our remote application removal feature, `http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html`
6. Google in-app billing, `http://developer.android.com/guide/market/billing/index.html`
7. Google license verification library, `http://developer.android.com/guide/publishing/licensing.html`
8. Hadoop, `http://hadoop.apache.org/`
9. Hash functions, `http://www.cse.yorku.ca/~oz/hash.html`
10. In-app billing, `http://developer.android.com/guide/market/billing/index.html`
11. Mobile threat report, `https://www.mylookout.com/mobile-threat-report/`
12. Number of available android applications, `http://www.appbrain.com/stats/number-of-android-apps/`

13. Proguard, http://developer.android.com/guide/developing/tools/proguard.html
14. Up to a million android users affected by malware, says report, http://www.linuxfordevices.com/c/a/News/Lookout-malware-report-2011/
15. Update: Security alert: Droiddream malware found in official android market, http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/
16. Freemarket: Shopping for free in android applications. Extended Abstract, to appear NDSS (2012)
17. Baker, B.S., Manber, U.: Deducing similarities in java sources from bytecodes. In: Proceedings of the USENIX Annual Technical Conference (1998)
18. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of MobiSys (2011)
19. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification. John Wiley and Sons (2000)
20. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of ACM CCS (2011)
21. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 321–330. ACM, New York (2008)
22. 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)
23. Hu, X., Cker Chiueh, T., Shin, K.G.: Large-scale malware indexing using function call graphs. In: Proceedings ACM CCS (2009)
24. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: Feature hashing malware for scalable triage and semantic analysis. In: Proceedings of ACM CCS (2011)
25. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of ICSE (2007)
26. Weinberger, K., Dasgupta, A., Langford, J., Smola, A., Attenberg, J.: Feature hashing for large scale multitask learning. In: Proceedings of ICML (June 2009)
27. Kim, H., Jung, Y., Kim, S., Yi, K.: Mecc: memory comparison-based clone detector. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 301–310. ACM, New York (2011)
28. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. Journal of Machine Learning Research 7 (December 2006)
29. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering 32(3) (2006)
30. Schleimer, S., Wilkerson, D., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proceedings of the ACM SIGMOD/PODS Conference
31. Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., Strehl, A., Vishwanathan, V.: Hash kernels. In: Proceedings of AISTATS 2009 (2009)
32. Walenstein, A., Lakhotia, A.: The software similarity problem in malware analysis. In: Proceedings of Duplication, Redundancy, and Similarity in Software (2007)
33. Yarow, J., Terbush, J.: Android is totally blowing away the competition, http://www.businessinsider.com/chart-of-the-day-android-is-taking-over-the-smartphone-market-2011-11
34. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Droidmoss: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (2012)

# ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems

Min Zheng, Patrick P.C. Lee, and John C.S. Lui

Dept of Computer Science and Engineering, The Chinese University of Hong Kong
{mzheng,pclee,cslui}@cse.cuhk.edu.hk

**Abstract.** With the rising threat of smartphone malware, both academic community and commercial anti-virus companies proposed many methodologies and products to defend against smartphone malware. Thus, how to assess the effectiveness of these defense mechanisms against existing and unknown malware becomes important. We propose *ADAM*, an automated and extensible system that can evaluate, via large-scale stress tests, the effectiveness of anti-virus systems against a variety of malware samples for the Android platform. Specifically, ADAM can automatically transform an original malware sample to different variants via repackaging and obfuscation techniques in order to evaluate the robustness of different anti-virus systems against malware mutation. The transformation and evaluation processes of ADAM are *fully automatic*, *generic*, and *extensible* for different types of malware, anti-virus systems, and malware transformation techniques. We demonstrate the efficacy of ADAM using 222 Android malware samples that we collected in the wild. Using ADAM, we generate different variants based on our collected malware samples, and evaluate the detection of these variants against commercial anti-virus systems.

## 1 Introduction

Malware (e.g., worms, viruses, and trojans) has been a well-known threat in the computing and networking communities. With the proliferation of smartphones, the threat of smartphone malware becomes more formidable. TGDaily [49] reported that there was a 33% increase in smartphone malware over 2009. As of October 2011, Tencent Mobile Security Laboratory [48] identified around 13,000 and 6,000 mobile phone viruses in the Symbian and Android platforms respectively. Given the threat of smartphone malware, researchers (e.g., [10,11,13,35,42,44,46,52]) have proposed various smartphone malware detection systems, and anti-virus software companies also develop commercial security solutions to detect smartphone malware. However, new pieces of smartphone malware keep evolving and attacking various distributions of smartphone platforms [36]. Thus, understanding the smartphone malware battle between the good and evil sides is critical for the community to improve the state of the art of the smartphone malware detection solutions. This motivates us to design a system that can *stress test* an anti-virus solution, so that one can systematically evaluate the effectiveness (or ineffectiveness) of existing smartphone malware detection systems against the emergence of smartphone malware.

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 82–101, 2013.

Evaluating smartphone malware detection is a non-trivial issue, especially with the challenge that there are a wide variety of smartphone operating systems available nowadays. In this work, we focus on *Android*, a Linux-based operating system that runs Java-based applications. Android applications can be directly self-signed and published by application developers through the official Android Market [6] without being subject to any official security validations. This unmoderated nature of Android provides a fertile ground for the development of both benign and malicious applications. As reported by International Data Corporation [30], Android led all smartphone OSes with 38.9% of market share in 2011, and is expected to grow to more than 40% of the market through 2015. Meanwhile, Android also becomes the most targeted operating system for smartphone malware [36]. Note that even the Android Market applies stringent security checks to its hosted applications, it cannot entirely resolve the malware distribution among Android phones, since some countries may ban the access to the Android Market (see Section 6). Thus, by focusing on the Android platform, our evaluation study can provide representative insights into the robustness of existing smartphone malware detection systems.

There are number of studies (e.g., [14,15,37,38]) that focus on evaluating the effectiveness of existing malware detection systems. Such evaluation studies employ different *obfuscation* techniques to transform a malware program into different variants (with the original malicious behavior preserved), and then check whether a malware detection system still treats the variants as malware. Note that most of these studies (e.g., [14,15,38]) focus on PC-based malware only, and limited studies (e.g., [37]) consider malware for smartphones. Given the growing popularity of smartphones, there is an urgent need to understand the effectiveness of anti-virus systems on smartphones, as well as their robustness against new and evolving malware. Furthermore, it remains challenging to scale up the evaluation to a large number of malware samples, as we need to ensure the correctness of various obfuscation techniques for each malware sample. Although we narrow down our focus on Android, the evaluation is still overwhelmed by numerous Android malware samples in the wild [48] as well as various malware detection solutions. Thus, the key motivation of this work is to develop an evaluation system that can *automatically* apply to general classes of smartphone malware and anti-virus solutions, and ultimately, support large-scale evaluation.

In this paper, we design and implement *ADAM*, an automated system for evaluating the detection of Android malware. ADAM applies different transformation techniques to generate different variants of each Android malware sample, and evaluates the effectiveness of different smartphone malware detection systems in identifying such malware variants. ADAM is designed to be *automated*, *generic*, and *extensible*. It automatically transforms an Android malware sample into different variants through various repackaging and obfuscation techniques, while preserving the original malicious behavior. ADAM then evaluates the detection of these variants against different smartphone malware detection systems. Such malware transformations and detection evaluations are generic enough to support heterogeneous malware samples and malware detection systems, respectively. Lastly, ADAM can be extensible to support new implementations of malware transformations and detection evaluations.

As a proof of concept, we demonstrate how ADAM can be used to assess the robustness of existing anti-virus systems in practice. We collected 222 malware samples in the wild. We use ADAM to generate different variants for each collected malware sample, and show that ADAM has a very high success rate in the automated generation of variants. We proceed to pass the variants to different commercial anti-virus engines hosted on the web portal VirusTotal [51]. We discuss the findings and implications based on the detection results returned from VirusTotal, but we emphasize that ADAM can also be integrated with other anti-virus systems.

The rest of the paper proceeds as follows. In Section 2, we provide a brief background on how to prepare and generate an Android application. In Section 3, we present the design of ADAM. In Section 4, we present various transformation techniques to generate different malware variants. In Section 5, we present our evaluation results against different anti-virus systems. In Section 6, we discuss several open issues. Section 7 surveys related work, and Section 8 concludes the paper.

## 2 Background

Let us describe the software life cycle of building an Android application from source code, as well as the reverse engineering process of an Android application. This lays the foundation of how our ADAM system transforms an Android malware application into another runnable Android malware variant while preserving the malicious behavior.

### 2.1 Building an Android Application

An Android application is mainly written in Java source code. The build process of an Android application is to compile and package a Java source code project into an `.apk` file that can run on a smartphone device or emulator. We now summarize the key steps of the build process [2] as follows.

1. **Preparation.** An Android project contains Java source code (and possibly some other native code), as well as metadata such as resources and programming interfaces. The build process first converts the metadata information into Java code or interfaces.
2. **Compilation.** All Java source code files as well as the converted metadata are compiled together into `.class` files, which contain Java bytecode.
3. **Bytecode Conversion.** All Android applications run on the *Dalvik Virtual Machine (DVM)*, which is a runtime environment similar to the Java Virtual Machine (JVM) but is designed for mobile devices that generally have limited hardware resources. The build process converts all `.class` files into `.dex` files, which contain the Dalvik Executable bytecode.
4. **Building.** All resource files, including both non-compiled and compiled files, as well as the `.dex` files are then packaged (i.e., zipped) into a single `.apk` file.
5. **Signing.** The `.apk` file needs to be digitally signed before it can be published in well-known sites (e.g., Google Market). It is typical that the `.apk` file is signed with the application developer's private key, rather than by a centrally trusted authority [4]. As described in Section 1, this type of unmoderated mechanism leads to

proliferation of Android applications, but at the same time, allows easy penetration of malware programs.

6. **Alignment.** To optimize the performance of the Android program (e.g., reducing memory usage), the `.apk` file can be aligned along the byte boundaries with the zipalign tool [5]. Note that some integrated development environment (IDE), such as Eclipse with the ADT plugin, will automatically zipalign the `.apk` file after signing the file with the developer's private key.

### 2.2 Process of Reverse Engineering an Android Application

In order to stress test the effectiveness of an anti-virus system, we need to create a library of malware variants and from existing malware. In most cases, the source code of malware (or an Android application) is not readily available, but instead, we can only access its `.apk` file and its underlying `.dex` files. To generate various malware variants, one has to resort to *reverse engineering*. We review two approaches that can be used to reverse-engineer an Android application.

1. **Decompiling.** The goal of the decompiling process is to convert a `.dex` file (with the DVM bytecode) into the `.java` source code files. A typically approach is to first convert the `.dex` file to `.class` files (e.g., using the dex2jar utility [20]), which are then converted to `.java` files using Java decompiler (e.g., using the Java Decompiler utility [33]). It is important to note that, the decompiling process may generate a source code file that is significantly different from the original one.
2. **Disassembling.** The disassembling process is to convert a `.dex` file into `.smali` files (e.g., using utility like apktool [9]), which contain assembly-like code for the Android OS. The process takes the Dalvik opcodes of a `.dex` file and converts them into low-level instructions. Typically, the decoded `.smali` files can be rebuilt again back to a `.dex` file.

In this paper, we focus on using the disassembling approach to reverse-engineer an Android malware sample. Through the disassembling approach, we can systematically locate specific assembly-like instructions for different malware samples, and apply code obfuscation to generate malware variants. We elaborate this in Section 4.2.

## 3 Design Overview of ADAM

In this section, we present an overview on the design of ADAM, an automated system for evaluating the detection of existing Android-based malware detection systems.

### 3.1 Design Goals

ADAM aims for the following design goals:

– *Security analysis.* ADAM checks whether an Android-based malware sample in `.apk` format can be detected by an existing anti-virus system. For this analysis, we do not need the source code of the malware sample.

– *Automated transformation.* ADAM automatically transforms a malware sample into different malware variants, while preserving the original malicious behavior. No manual modification of a malware sample is required.
– *Generic application.* ADAM can be applied for general classes of Android-based malware samples and malware detection systems.
– *Extensibility.* ADAM provides an interface that can easily integrate new implementations of transformation techniques and detection methodologies.

### 3.2 Building Blocks

ADAM is composed of different building blocks. Figure 1 illustrates how different building blocks are involved in testing malware samples against anti-virus systems. Let us now describe how each building block works, and argue how the building blocks can be extended for different variants of implementation.

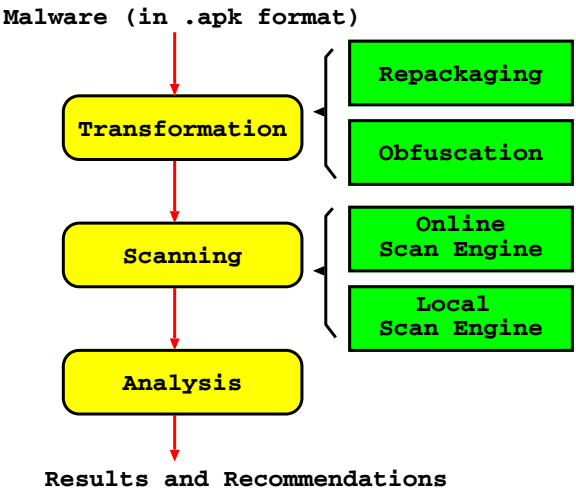



**Fig. 1.** Design flow of ADAM

**Transformation.** Given an input `.apk` malware file, ADAM transforms it into different variants of `.apk` files based on various transformation techniques, such that each output `.apk` file preserves the original malicious behavior of the input `.apk` files. We implement two classes of transformation techniques: *repackaging* and *code obfuscation*. Details of these techniques are described in Section 4. We emphasize that ADAM is extensible in the sense that one can plug-in other transformation techniques to generate different `.apk` variants.

**Scanning.** For each malware variant that we create, we pass it to an anti-virus scan engine. Here, we focus on the scan engines of commercial vendors, while we can also plug-in other malware detection systems with the correct interface.

In ADAM, we support two types of scan engines: *online* and *local*. An online scan engine refers to a web service that provides a library of open APIs. Users can upload an `.apk` file to the web service and obtain the results via the APIs. Typically, the web service is free of charge, but *rate-limit* the number of samples that can be scanned. Also, the scanning performance varies depending on the current network conditions. In our implementation, we use the VirusTotal web portal [51], which is connected to various commercial anti-virus systems at its backend. On the other hand, a local (or desktop) scan engine uses the command-line interface provided by an anti-virus vendor. It simply specifies an `.apk` file as an input command-line argument and obtains the scanned results. In our implementation, we integrate the Linux desktop version of the anti-virus engine obtained from Antiy [8]. Our evaluation study covers both online and local scan engines (see Section 5).

**Analysis.** ADAM collects the results from the anti-virus scan engines of different commercial vendors. One can determine if a scanned `.apk` file is a malware sample based on the decisions of one or multiple anti-virus systems. Aggregating the results of multiple anti-virus systems can potentially increase the detection rate[40]. The analysis results can be summarized and presented, so as to provide recommendations for anti-virus vendors to evaluate the effectiveness of the state of the art of malware detection for Android.

## 4 Malware Transformation

In this section, we present techniques that we use to transform a given malware sample into different variants. The resulting variants will be used by ADAM as inputs to evaluate the effectiveness of different malware detection systems. Specifically, we consider two classes of transformation techniques: *repackaging* and *code obfuscation*, both of which take an `.apk` file as an input and generate a different `.apk` file as an output. Furthermore, we require that the output of an `.apk` file preserves the same logic and functionality as the original input `.apk` file.

It is important to note that *by no means do we claim our transformation techniques are new*, as they have also been studied in other evaluation systems for malware detection (e.g., [14,15,37,38]). On the other hand, ensuring the applicability of existing transformation techniques in *general* Android applications remains a challenging issue. In the following, we consider a number of transformation techniques that can be *automated* (i.e., without manual intervention) for general `.apk` files. Hence, one can easily generate malware variants for a large number of malware samples.

### 4.1 Repackaging

In ADAM, we consider different repackaging methods that work directly on an input `.apk` file and regenerate a different `.apk` file *without modifying the source code of the input* `.apk` *file*. Thus, making the transformation easily deployable and preserving the functionality of the input `.apk` file. We consider three techniques that are currently supported by ADAM. One common key feature of all such techniques is that they are

all built on the official Android or Java development utilities, which we expect are more robust and stable than other third-party tools.

**Alignment.** The alignment technique realigns the data of an `.apk` file, so as to generate different content but preserving the same logic for an `.apk` file. We use zipalign[5] to realign the uncompressed data within an `.apk` file (e.g., images or raw files) on 4-byte boundaries so that all portions can be accessed directly via `mmap()` function. The zipalign utility is available in the Android SDK, and is originally designed for providing optimization for `.apk` files. Since the alignment optimization changes the internal structure of the `.apk` file, it accordingly changes some of the signature patterns, such as the cryptographic hash of the `.apk` file. If an anti-virus system directly identifies malware based on the cryptographic hash signature (e.g., MD5), then the alignment technique can easily evade the detection of anti-virus system.

Our system applies alignment to an `.apk` file as follows. It is recommended [5] that all `.apk` files are aligned on 4-byte boundaries to achieve optimization. Thus, it is possible that the original input `.apk` file has already been 4-byte aligned. To ensure that the `.apk` file is actually transformed to a different output, our current implementation applies zipalign with the 8-byte alignment boundaries.

**Re-sign.** The re-sign technique is to generate a different signature for an `.apk` file. Android requires that every `.apk` file be digitally signed before the `.apk` file can be published and run on a smartphone. According to the official documentation [4], it is allowed and typical that an `.apk` file is self-signed by its application developer *without involving a trusted central authority*. One of our observations (which is not officially documented) is that an `.apk` file can be re-signed multiple times with different certificates and private keys, so that a different signature is generated and attached to the `.apk` file. This can evade the detection of anti-virus systems that simply identifies malware by its original `.apk` signature.

To (re-)sign an application, we use the Keytool and Jarsigner utilities, both of which are available in the Java SDK [41]. We first use Keytool to generate a self-signed key and put the key in a key store. We then use Jarsigner to sign an `.apk` file using the key store as the input.

**Rebuild.** The rebuild technique disassembles an `.apk` file and rebuilds the assembly code (without being modified) into another `.apk` file. We use apktool [9] to disassemble the Dalvik bytecode within an `.apk` file into Smali code [34] (see Section 2), and rebuild the Smali code back to Dalvik bytecode using apktool. After the disassemble process, the original `.apk` file and the repackaged `.apk` file are exactly the same, but repackaged `.apk` file will have different Dalvik bytecode order from the original one depend on the parser's analysis. This makes the resulting Dalvik bytecode (and hence the `.apk` file) different from the original one and at the same time, preseves the logic and functionality of the original `.apk` file.

We then sign the output `.apk` file with a randomly generated private key as described in the re-sign technique. This rebuild technique is effective to evade anti-virus systems that use cryptographic hash and/or `.apk` signature for malware identification.

### 4.2 Code Obfuscation

In code obfuscation, we modify the program code of an `.apk` file so as to make an anti-virus system more difficult to reverse engineer[17]. In particular, code obfuscation changes the size and content of the `.apk` file, but without modifying the logical behavior. Our code obfuscation techniques operate on Smali code [34], an assembly-like language based on the Dalvik executable code. The Smali syntax provides the debug information of how the variables and methods are invoked. This enables us to easily add obfuscated code to an `.apk` file.

To apply code obfuscation to an `.apk` file (of a malware sample), we first disassemble it using the apktool utility [9] into a `.smali` file. We modify the `.smali` file according to each obfuscation technique which we will describe shortly. We then rebuild the modified `.smali` file into an `.apk` file using apktool and sign the output `.apk` file, as in our previously proposed rebuild and re-sign techniques, respectively (see Section 4.1). One can use zipalign [5] for data alignment so to generate an optimized `.apk` file.

There are various code obfuscation techniques proposed in the literature, especially for the Java language (e.g., see [17]) on which Android is based. Note that some code obfuscation techniques depend on the underlying semantics of a program, and typically require *manual code modification* that cannot be easily automated. For example, substitution of code with different lines of code may need to be carefully carried out so as to preserve the original malicious behavior [37]. Also, our goal is to show that even with simple obfuscation techniques, one can generate new malware samples that can easily evade the detection of anti-virus systems. Thus, we consider several general code obfuscation techniques *that can be automated*, while being sufficient to subvert most of the anti-virus systems.

**Inserting Defunct Methods** (e.g., [14,15]). We add new methods that perform defunct functions to Smali code, and these inserted methods do not change the logic of the original source code. The rationale of this obfuscation technique is to modify the *method table* in the Dalvik bytecode, and hence change the signature that is generated based on the method table.

There are many ways to add defunct methods. In our implementation, we implement a `Log.d` debug method [3] that prints a simple string in Android (obviously, other defunct methods can be added to ADAM). We first disassemble the method into Smali format. We then insert the `Log.d` method before the constructor method of each class in the disassembled `.smali` file. To locate a constructor method, we search for the string "`# direct methods`" in each `.smali` file, because the constructor method must follow this string. Figure 2 shows how we insert a defunct method.

**Renaming Methods** (e.g., [37]). We obfuscate a method name with a different string, and hence change the signature that is generated by the method name. In our implementation, we first identify all the system library method names from `Android.jar` in Android SDK, so as to differentiate them from user-defined methods. Then we search for all user-defined methods (i.e., other than the library methods) in each `.smali` file, and append a randomly generated string (e.g., "`abc10`") at the end of each user-defined method that we find. We modify the method name when the method is first defined, as

```
...
# direct methods
.method public OFLog(Ljava/lang/String;)V
...
.method public constructor <init>()V
...
```

**Fig. 2.** Inserting a defunct method (i.e., `OFLog`)

well as when it is called within the code. We point out that our system can also rename other types of identifiers, including packages, variables, and classes, so as to make the code more obfuscated. For example, Figure 3 shows how we rename a user-defined method "`foo`" into "`fooabc10`". To summarize, this type of obfuscation can evade anti-virus system that uses method names to generate virus signatures.

```
.method public static fooabc10(Ljava/lang/String;)V
...
invoke-static {v1}, Lcom/test;->fooabc10(Ljava/lang/String;)
```

**Fig. 3.** Renaming a method from `foo` to `fooabc10`

**Changing Control Flow Graphs (CFGs).** Some anti-virus systems (e.g., Androguard [1]) can use CFGs to generate signatures and detect the presence of malware. A CFG signature can be defined based on a grammar table [12]. Here, we modify the CFG of a `.smali` file and so as to change its CFG signature.

We consider one particular CFG obfuscation called the *Goto-obfuscation*. We insert `goto` statements to each method in a `.smali` file. At the beginning of a method, we insert a `goto` statement to jump to the end of the method; at the end of the method, we insert another `goto` statement to return to the beginning of the method. We insert a `return` statement before the second `goto` statement, so that the latter will not be called again when the method is finished. Figure 4 illustrates how we insert `goto` statements into a method `foo`.

```
.method public foo(Ljava/lang/String;)V
 .prologue
 goto :CFGGoto2
 :CFGGoto1
...
 return-void
 :CFGGoto2
 goto :CFGGoto1
.end method
```

**Fig. 4.** Goto-obfuscation

**Encrypting Constant Strings.** We encrypt all constant strings that we find in a `.smali` file, and decrypt them when they are being processed. This modifies the signatures that are generated by these constant strings. Here, we consider a simple symmetric encryption method based on the Caesar cipher, in which we shift the character byte of each alphabet letter (i.e., `A-Z` or `a-z`) by a constant integer value. For example, we can encrypt a string "`DecryptString`" in a TextView control by subtracting all bytes by 10. The encrypted string will become "`:[YhofjIjh_d]`". We then add the decryption method `decrypt` (i.e., by adding all bytes by 10) before the TextView control is called. Figure 5 shows how the example works. In summary, this type of obfuscation can evade anti-virus system that uses constant string to generate virus signature.

```
#direct methods
.method public static DecryptString\
(Ljava/lang/String;)Ljava/lang/String;
...
const-string v1, ":[YhofjIjh_d]"
...
invoke-static { v1},\
Lcom/test;->DecryptString\
(Ljava/lang/String;)Ljava/lang/String;
move-result-object v1
invoke-virtual {v0, v1}, Landroid/\
widget/TextView;->setText\
(Ljava/lang/CharSequence;)V
```

**Fig. 5.** Encrypting a constant string

## 5 Evaluation of Anti-virus Systems

In this section, we use ADAM to evaluate the effectiveness of current commercial anti-virus systems in the detection of Android smartphone malware, and examine their robustness of dealing with malware variants as stated in our previous section. We conduct large-scale analysis using ADAM as follows. We collect a total of 222 Android malware samples in the wild, and generate different variants for each collected malware sample based on our transformation techniques (see Section 4). We feed these malware variants into different commercial anti-virus systems, and test if these variants are diagnosed as malware. Our analysis provides us a comprehensive picture of the effectiveness of current commercial anti-virus systems. Most notably, it enables us to validate the *automated* and *generic* properties of ADAM in experimenting with large-scale malware samples and anti-virus systems.

### 5.1 Malware Dataset

We collect a total of 222 *distinct* Android malware samples (with unique MD5 hashes) from three different sources, including:

- **Old Public Samples.** We download 57 Android malware samples from [18], a well-known blog website that maintains a collection of mobile malware. The blog author frequently updates the blog and shares the most recent malware samples to the public. The 57 samples are the public Android malware samples that are published from March 2011 to September 2011. Given the well-publicized blog, current anti-virus systems should have a very high detection rate on these malware samples.
- **New Public Samples.** On Oct. 22, 2011, the blog [18] published 96 new Android samples from an anonymous source. We believe it is interesting to study how fast the anti-virus systems add the signature of these new malware samples to their databases.
- **Private Samples.** On Oct. 20, 2011, we obtained 69 Android malware samples from Antiy [8], an anti-virus company based in China. These samples are *unpublished*, so other anti-virus companies may not have enough signatures to detect these malware samples. Our hypothesis is that existing anti-virus systems will have a lower detection rate on these samples.

We carefully investigate the malicious logic of each of the 222 malware samples. We group the malware samples that have the same logic into a *family*. After our investigation, we group the 222 malware samples into 38 different families. Furthermore, we can classify the malware families into four categories, as we briefly describe below.

**Repackaging Malware.** All malware samples in this category are transformed from legitimate applications via the disassembling approach (see Section 2.2). Briefly speaking, an attacker adds extra permissions and malicious services to a legitimate application, and repackages everything into a new (malicious) application. When a user installs the repackaged application, the application will perform malicious activities such as collecting the user's personal information and sending it to a remote server, or sending payment short messages to some premium SMS numbers. In our malware dataset, we have 138 malware samples from 12 families that fall into this category. For example, there are 32 samples of a family called Geinimi. All the Geinimi samples are transformed from different legitimate applications. Each of these malware samples has a common Java package called Geinimi, which contains a service called Adservice that performs malicious activities as listed above. This service modifies `AndroidManifest.xml` in the `.apk` file and starts automatically when the system boots up. In this category, there are also some well-known malware families such as DroidKungFu, BaseBridge, and Hongtoutou that have been studied in the literature [25].

**Display-Modification Malware:** This type of malware has a feature that all malware samples have the same application structure and same malicious behavior, but they have different icons, names, wallpapers, themes, or pictures. We have 46 malware samples of two families in our malware dataset that belong to this category. For example, the Kmin family is a wallpaper changer application, and contains 42 samples in our dataset.

**Camouflage Malware.** This category of malware has a key feature that they imitate the same user interface of some original application in order to steal a user's account credentials. We have a total of 13 samples of six families in our malware database. For example, one family is called FakeNetflix, whose package name is "com.netflix.mediaclient"

and is the same as that of the legitimate application Netflix. This family of malware displays a login screen to the user so as to steal the user's password and send it to a remote server.

**Generic Malware.** This type of malware is just a plain malicious application without being camouflaged as any legitimate application. An attacker may simply physically access a smartphone and install the malware there. We have 25 malware samples of 18 families in our malware dataset under this category. For example, one family is called NickiSpy, whose package name is called "com.nicky.lyyws.xmall". It can steal users' credentials and wiretap users' phone calls in the background. It can also record any phone conversation and store it under the directory named "shangzhou/callrecord" in the SD card.

### 5.2 Anti-virus Systems

We conduct our evaluation against the commercial anti-virus products hosted on the web portal VirusTotal [51] (see Section 3.2) in October and November 2011. Note that VirusTotal hosts over 40 anti-virus products, and our study only focuses on the *top 10* products that give the highest detection rates for our 222 original malware samples (i.e., without transformations) in November 2011.

In addition, in February 2012, we also evaluate a commercial anti-virus product that we obtained from Antiy [8], and the product is known to run the same engine as that being deployed in smartphone platforms.

We note that some anti-virus systems, such as Androguard [1], can detect malware based on control-flow-graph signatures (see Section 4.2). However, our evaluation does not consider Androguard, whose latest version is released in September 2011 at the time of this paper being written, while our malware samples are collected since October 2011. We think that it is unfair to evaluate Androguard using the malware samples collected after its latest release.

### 5.3 Analysis

For the 222 malware samples we collected, we apply our transformation techniques stated in Section 4, including three repackaging techniques and four code transformation techniques, to each malware sample. All samples can be successfully transformed by the Re-sign technique. However, two of the samples can be transformed by the Alignment technique, but cannot be transformed by the Rebuild and the four code transformation techniques because of the re-compilation errors. Also, 10 of the samples cannot be transformed by all techniques except Resign, mainly because they just contain `.dex` files that cannot be rebuilt into `.apk` files. Therefore, we can only generate a total of 1484 variants. Nevertheless, our transformation techniques have a success rate of 95.5%, showing the robustness of ADAM.

### 5.4 Results

We evaluate all malware samples and their transformation variants against different anti-virus systems.

**Table 1.** Detection rates for various anti-virus systems: Time: November 2011

| AV Products | Original | Alignment | Re-sign | Rebuild |
|---|---|---|---|---|
| **Kaspersky** | 95.95% | 94.34% | 94.59% | 94.76% |
| **F-Secure** | 95.50% | 95.75% | 95.05% | 91.90% |
| **Emsisoft** | 94.59% | 93.87% | 93.69% | 75.24% |
| **Ikarus** | 94.59% | 94.34% | 93.69% | 75.24% |
| **GData** | 94.14% | 93.87% | 93.69% | 90.95% |
| **TrendMicro** | 94.14% | 91.98% | 92.79% | 77.62% |
| **NOD32** | 92.79% | 88.68% | 88.29% | 95.24% |
| **Sophos** | 92.79% | 94.81% | 94.14% | 78.10% |
| **Antiy-AVL** | 92.34% | 91.98% | 89.19% | 72.38% |
| **Fortinet** | 90.99% | 89.15% | 88.74% | 71.43% |
| **Overall Average** | 93.78% | 92.88% | 92.39% | 82.29% |

(a) Detection of original malware samples and their variants generated by repackaging.

| AV Products | Insert | Rename | Change CFG | Str. Encrypt |
|---|---|---|---|---|
| **Kaspersky** | 93.81% | 73.33% | 94.76% | 90.95% |
| **F-Secure** | 90.00% | 90.00% | 90.48% | 68.57% |
| **Emsisoft** | 83.81% | 26.67% | 82.86% | 25.24% |
| **Ikarus** | 83.81% | 26.67% | 83.33% | 25.24% |
| **GData** | 90.95% | 90.48% | 91.43% | 88.10% |
| **TrendMicro** | 61.90% | 61.90% | 63.81% | 35.71% |
| **NOD32** | 95.24% | 91.90% | 95.24% | 90.48% |
| **Sophos** | 54.29% | 54.29% | 54.76% | 49.05% |
| **Antiy-AVL** | 70.00% | 19.05% | 67.14% | 19.52% |
| **Fortinet** | 48.57% | 15.71% | 42.86% | 16.67% |
| **Overall Average** | 77.24% | 55.00% | 76.67% | 50.95% |

(b) Detection of malware variants generated by code obfuscation.

**(1) Analysis of All Transformation Techniques.** Table 1 shows the experimental results of the detection rates of each of the top 10 anti-virus systems that we choose, while the evaluation was conducted on November 21, 2011. We discuss our findings below.

• *Original malware samples.* We first test the original malware samples that we collected (without applying any transformation). The top 10 anti-virus systems we consider performed well in the original sample detection, they all have over 90% of detection rates. The average detection rate is 93.78%. This indicates that anti-virus companies have already begun to value the security of Android systems, and responded quickly to the emergence of new Android malware. In the following, we analyze the detection rates due to different transformation techniques when compared to the detection of the original malware samples.

• *Alignment.* As mentioned in Section 4, alignment via zipalign only changes the cryptographic hash signature of an `.apk` file. After alignment, there are only slight drops of the detection rates for all anti-virus systems (by at most 4%).

• *Re-sign.* We re-sign the `.apk` file of each malware with a random key. We observe that the detection rates of all the 10 anti-virus systems that we consider are not much different from the results for the original samples and the alignment transformation. We believe the reason is that most anti-virus products apply the unzipping process to deal with the `.apk` files before scanning. The re-signed `.apk` file will be no different from the original `.apk` file after the unzipping process because the signature process is based on the whole `.apk` file, and does not change the content of an `.apk` file. Note that after being re-signed, each `.apk` file has a different cryptographic hash. This may reduce the detection rate if an anti-virus system relies on cryptographic hashes as signatures, similar to the observations in the alignment transformation.

• *Rebuild.* After the rebuild process, the average detection rate of the 10 anti-virus products drops from 93.78% (in the original sample detection) to 82.29%. To understand this phenomenon, we used Dedexer [50] and UltraCompare [31] to analyze the original samples and the rebuilt variants. Dedexer is a disassembler tool for `.dex` files. Unlike apktool, the Dedexer tool can be used as a `.dex` parser to generate a detailed log file on the internal structure of a `.dex` file. UltraCompare is a comparison utility that can handle binary file comparison, text comparison and folder comparison.

After rebuilding a `.dex` file, the result shows that the `.dex` file has changed. We use UltraCompare to compare the detailed log files of these `.dex` files. We find out that the checksum, the signature, some offsets, and some size values have changed. In addition, although the strings or method names do not change, their index orders are different from the original `.dex` file. These changes imply that just using fragments of a `.dex` binary file as the malware signature may not be effective, and it may reduce the detection rate.

• *Insert defunct methods.* After the insert defunct methods transformation, the average detection rate of the 10 anti-virus systems has decreased from 93.78% down to 77.24%. Then we use UltraCompare to compare the detailed log files of the original samples and malware variants. The most distinctive difference between the rebuilt variant and the insert-defunct-methods variant is the method table. In the method table, the total number of methods has changed and the size of the method table becomes bigger because we insert additional defunct method implementations. Therefore, if an anti-virus system uses the hash value of all of the method names as the signature, then adding defunct methods will make the detection ineffective. For example, we insert defunct methods process to one of our malware samples called `snake`. Then we compare the variant with its original sample. We find that the total number of methods increases from 239 to 259, and the file size is only increased by 4%. Again, these changes are due to the insertions of defunct code.

• *Renaming methods.* After the renaming methods transformation, the average detection rate has decreased from 93.78% down to 55.00%. In particular, the detection rates of some anti-virus products drop significantly, for example, from 92.34% to 19.05% for Antiy-AVL. In addition to the changes in the rebuild process, the method table has also

**Table 2.** Detection rates for various anti-virus systems using original malware samples and their variants generated by the repackaging techniques: October 2011

| AV Products | Original | Alignment | Re-sign | Rebuild |
|---|---|---|---|---|
| **F-Secure** | 93.24% | 95.28% | 94.59% | 89.05% |
| **Kaspersky** | 93.24% | 90.09% | 89.64% | 62.38% |
| **Emsisoft** | 90.99% | 90.09% | 87.84% | 61.90% |
| **Ikarus** | 90.99% | 90.09% | 87.84% | 61.43% |
| **GData** | 88.74% | 92.45% | 91.44% | 86.67% |
| **Sophos** | 88.74% | 86.32% | 86.49% | 68.10% |
| **Antiy-AVL** | 86.04% | 75.00% | 73.42% | 54.76% |
| **TrendMicro** | 85.59% | 75.94% | 74.32% | 53.81% |
| **Fortinet** | 79.28% | 68.87% | 68.47% | 43.33% |
| **NOD32** | 77.93% | 55.66% | 52.25% | 35.24% |
| **Overall Average** | 87.48% | 81.98% | 80.63% | 61.67% |

changed significantly. Also, the implementation of methods has also changed because the methods now invoke different method names. This indicates that if anti-virus systems use method names to generate signatures, then they may fail in the detection. We observe that the renaming method transformation is more effective in evading malware detection compared to inserting defunct methods.

• *Changing control flow graphs.* We used the Goto obfuscation technique so that every method implementation has been added with 4 lines of Goto statement while other changes are the same as the rebuild process. We find out that the result is similar to that of the insert-defunct-methods transformation. The average detection rate has decreased from 93.78% down to 76.67%.

• *String encryption.* After this transformation process, the string table and method table will change because of the string encryption and the insertion of decryption methods. The average detection rate of all anti-virus systems has decreased from 93.78% to 50.95%. This indicates that a lot of anti-virus systems that we consider use constant strings as the signature to detect the presence of malware.

**(2) Evolution of Malware Detection.** We used ADAM to carry out the *first* stress test on all anti-virus systems in October 2011 right after we collected all malware samples. Here, we only focus on the detection of original malware samples and their variants generated by the repackaging techniques. Table 2 shows the detection rates of the top 10 anti-virus systems that we consider in Table 1. Compared with the detection rates on Table 1, which we carried out the experiment in November 2011, we see that most of the anti-virus systems *improve* in the malware detection, in particular, on the original malware samples. This shows that anti-virus systems are *rigorously updating* their signature databases. However, there are still a number of anti-virus systems which are not robust against simple malware transformations based on repackaging.

In February 2012, we obtained from Antiy [8] an anti-virus engine that runs atop a desktop PC with the Linux operating system and is known to have the same detection

**Table 3.** Detection rates of the Antiy's anti-virus product in February 2012

| Original | Alignment | Re-sign | Rebuild |
|---|---|---|---|
| 94.59% | 96.69% | 94.59% | 95.23% |
| **Insert** | **Rename** | **Change CFG** | **Str. Encrypt** |
| 94.28% | 93.80% | 98.57% | 94.28% |

logic as that being deployed in smartphone platforms. We conduct evaluation against it in February 2012 using the same set of variants. Table 3 shows the results. We observe that the detection rates for the original malware samples and all their variants can achieve over 90%. This shows that commercial anti-virus products evolve to become more robust against malware transformations.

## 6 Discussion

In this section, we describe several open issues that we have not addressed in this work, and suggest the future directions.

**Signature Coverage.** While we show that our transformation techniques can make a malware application evade the detection of a number of commercial anti-virus systems, it is non-trivial to accurately infer the underlying signatures being used by such systems. Also, although we confirm that some companies use the same anti-virus engine for both desktop and mobile versions (see Section 5.2), we cannot verify if all anti-virus systems that we tested on VirusTotal apply the same detection logic as in their mobile versions, as the latter can be better. One future work is to apply ADAM to evaluate both desktop and mobile versions of an anti-virus product and compare their detection performance.

**Distribution Model.** We point out that it is generally difficult to distribute malicious applications through the official Android Market because of strict application checking. However, we believe that hackers can upload any malware to the third-party markets, given that the Android Market may be banned by some countries such as China [53]. Thus, users may have to use third-party markets to access mobile applications. In addition, "rooted" smartphones can install any applications and bypass any strict checking imposed by the Android OS, thereby making the spread of malware more feasible. It is interesting to further study the impact of the distribution model of mobile applications on the spread of malware.

**Defense Solutions.** We propose several solutions that can defend against obfuscation and repackaging techniques we discussed. First, one can use a `.dex` parser to extract signatures from a `.dex` file to counter common obfuscation methods, because the logic and functionality of the `.dex` file does not change. Second, a good optimizer can handle the insertion of defunct methods and changing control flow graph methods, because all redundant code can be eliminated after the code optimization method. Third, using fuzzy hashing [29] to detect unknown malware appears to be a promising approach, but how to find the optimized parameters to control the anchor points remains a challenging research problem.

## 7 Related Work

Smartphone malware (e.g., viruses, worms, and trojans) presents a critical security threat to smartphones. A piece of malware can reside in smartphones, perform malicious activities, and compromise the trusts of smartphones. As the first smartphone worm Cabir appeared in 2004 [47], the research community has been alerted about the severity of smartphone malware [19,26,32]. While smartphone malware first appeared in Symbian OS, Schmidt *et al.* [45] implemented the first malware for Android. Since then, there has been a rapid spread of malware in different mobile platforms including Android and iOS (see the survey of [25]). Recently, Schlegel *et al.* [43] demonstrated an Android malware called Soundcomber that can steal voice data with only limited permission privileges.

Existing commercial anti-virus solutions identify smartphone malware mainly based on static signature-based detection, which aims to identify any malicious patterns of the source code of an application without executing it. However, smartphones typically have scarce computational and bandwidth resources, and so it is ineffective to have smartphones deploy anti-virus solutions and update the latest signatures in a timely manner. SmartSiren [13] is a proxy-based, collaborative detection system that collects the activities from various smartphones in order to detect the existence of malware. Bose *et al.* [10] propose a machine-learning-based framework that detects the presence of smartphone malware by looking into malicious behavior signatures, and show that behavioral detection gives higher detection accuracy and is more resilient to code transformation than conventional signature-based detection. Schmidt *et al.* [44] consider a similar collaborative system as in [13] and use behavioral detection, with the emphasis on Android systems. Paranoid Android [42] uses remote servers to examine the replicas of Android phones and identify security threats. Crowdroid [11] is a behavioral detection malware system for Android, and collects the system-call traces of various real Android users to identify malware. In summary, the above approaches mainly use a network-based system that remotely runs the malware detection.

There are host-based malware detection systems that directly run on smartphones. Xie *et al.* [52] propose access-control defense to limit the accesses of malware to critical system resources. VirusMeter[35] identifies malware that causes excessive battery power consumption on mobile devices. Andromaly[46] is an Android-based malware detection system that applies machine learning to identify anomalous behavior.

A number of researchers (e.g., [27,39]) motivate the needs and challenges of testing security software, and AMTSO [7] is one major organization that propose different standards for testing anti-virus systems. There have been research studies that focus on testing the resilience of existing malware detection systems. Christodorescu and Jha [14,15] show that simple code obfuscation techniques suffice to evade the detection of commercial anti-virus systems, which are mainly built on static signature-based detection. Moser *et al.* [38] show that obfuscation techniques based on opaque constants can evade static detection systems that consider instruction semantics (e.g., [16]). Note that these studies mainly consider malware on PCs but not on mobile devices. Morales *et al.* [37] evaluate the resilience of commercial anti-virus systems for the Windows Mobile OS, and consider it only with two virus samples and four commercial

anti-virus systems. Our work, on the other hand, targets the Android OS and covers significantly larger sets of virus samples and commercial anti-virus systems.

There are studies that investigate the security issues in Android smartphones, such as privacy leakage and permission usage. Enck *et al.* [23] analyze the security of existing Android applications, by decompiling and recovering the Java source code of Android applications in Google's Android Market. Taintdroid [22] uses dynamic taint tracking to identify any privacy leakage in Android applications (a similar privacy leakage detection system PiOS [21] is designed for Apple iOS). AppFence [28] extends Taintdroid by controlling how private data can enter or leave an Android application. Stowaway [24] uses static analysis to identify the permission usage of the API calls in Android applications. Our work mainly focus on generating malware threats and examine the effectiveness of malware detection in Android smartphones.

## 8 Conclusions

We present ADAM, an automated, generic, and extensible platform that evaluates the detection of Android malware detection systems. ADAM applies different transformation techniques, including repackaging and code obfuscation, to an Android malware sample to generate different variants. Then it applies these variants to stress test the robustness of a wide range of anti-virus systems. ADAM is designed to be automatic, generic, and extensible for assessing the state of the art of Android malware detection. We conduct large-scale studies based on 222 Android malware samples against various commercial anti-virus systems, so as to demonstrate how ADAM provides recommendations to improve current detection mechanisms. Our ADAM prototype is available for download at: **http://ansrlab.cse.cuhk.edu.hk/software/adam**.

**Acknowledgments.** We would like to thank our shepherd, Lorenzo Martignoni, and the anonymous reviewers for their valuable comments.

## References

1. Androguard (2010), `http://code.google.com/p/androguard/`
2. Android. Android Developers - Building and Running, `http://developer.android.com/guide/developing/building/index.html`
3. Android. Log, `http://developer.android.com/reference/android/util/Log.html`
4. Android. Signing Your Applications, `http://developer.android.com/guide/publishing/app-signing.html`
5. Android. zipalign, `http://developer.android.com/guide/developing/tools/zipalign.html`
6. Android Market, `https://market.android.com/`
7. Anti-Malware Testing Standards Organization, `http://www.amtso.org`
8. Antiy, `http://www.antiy.net`
9. Apktool, `http://code.google.com/p/android-apktool/`
10. Bose, A., Hu, X., Shin, K.G., Park, T.: Behavioral Detection of Malware on Mobile Handsets. In: Proc. of ACM MobiSys (2008)

11. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: Behavior-Based Malware Detection System for Android. In: ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (2011)
12. Cesare, S., Xiang, Y.: Classification of Malware Using Structured Control Flow. In: Proc. of the Eighth Australasian Symposium on Parallel and Distributed Computing (2010)
13. Cheng, J., Wong, S.H., Yang, H., Lu, S.: SmartSiren: Virus Detection and Alert for Smartphones. In: Proc. of ACM MobiSys (2007)
14. Christodorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: Proc. of USENIX Security Symposium (2003)
15. Christodorescu, M., Jha, S.: Testing Malware Detectors. In: Proc. of ISSTA (2004)
16. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-Aware Malware Detection. In: IEEE Symposium on Security and Privacy (2005)
17. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report 148, Dept. of Computer Science, University of Auckland, New Zealand (July 1997)
18. Contagio Mobile, http://contagiominidump.blogspot.com/
19. Dagon, D., Martin, T., Starner, T.: Mobile Phones as Computing Devices: The Viruses are Coming! Pervasive Computing 3(4), 11–15 (2004)
20. dex2jar, http://code.google.com/p/dex2jar/
21. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: PiOS: Detecting Privacy Leaks in iOS Applications. In: Proc. of NDSS (2011)
22. Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proc. of USENIX OSDI (2010)
23. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proc. of USENIX Security Symposium (2011)
24. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: Proc. of ACM CCS (2011)
25. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A Survey of Mobile Malware in the Wild. In: Proc. of ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (2011)
26. Guo, C., Wang, H.J., Zhu, W.: Smart-Phone Attacks and Defenses. In: ACM SIGCOMM HotNets (2004)
27. Harley, D.: Making Sense of Anti-Malware Comparative Testing. Information Security Tech. Report 14(1) (February 2009)
28. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In: Proc. of ACM CCS (2011)
29. Hurlbut, D.: Fuzzy Hashing for Digital Forensic Investigators (May 2011), http://accessdata.com/downloads/media/Fuzzy_Hashing_for_Investigators.pdf
30. IDC. Worldwide Smartphone Market Expected to Grow 55% in 2011 and Approach Shipments of One Billion in 2015, According to IDC (June 2011), http://www.idc.com/getdoc.jsp?containerId=prUS22871611
31. IDM Computer Solutions, Inc. File Compare — UltraCompare Professional (2011) http://www.ultraedit.com/products/ultracompare.html
32. Jamaluddin, J., Zotou, N., Edwards, R., Coulton, P.: Mobile Phone Vulnerabilities: A New Generation of Malware. In: Proc. of IEEE Int. Symp. on Consumer Electronics (2004)
33. Java Decompiler, http://java.decompiler.free.fr/
34. JesusFreke. smali (2011), http://code.google.com/p/smali/

35. Liu, L., Yan, G., Zhang, X., Chen, S.: VirusMeter: Preventing Your Cellphone from Spies. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 244–264. Springer, Heidelberg (2009)
36. McAfee Labs. McAfee Threats Report: Second Quarter 2011 (2011)
37. Morales, J.A., Clarke, P.J., Deng, Y.: Testing and Evaluating Virus Detectors for Handheld Devices. Journal in Computer Virology 2(2), 135–147 (2006)
38. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In: Proc. of ACSAC (2007)
39. Muttik, I., Vignoles, J.: Rebuilding Anti-Malware Testing for the Future. In: Virus Bulletin Conference (2008)
40. Oberheide, J., Cooke, E., Jahanian, F.: CloudAV: N-Version Antivirus in the Network Cloud. In: Proc. of USENIX Security (2008)
41. Oracle. JDK Tools and Utilities (2010), http://download.oracle.com/javase/1.5.0/docs/tooldocs/#security
42. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Android: Versatile Protection For Smartphones. In: Proc. of ACSAC (2010)
43. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proc. of NDSS (2011)
44. Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yüksely, K.A., Camtepe, S.A., Albayrak, S.: Static Analysis of Executables for Collaborative Malware Detection on Android. In: Proc. of IEEE ICC (2009)
45. Schmidt, A.-D., Schmidt, H.-G., Batyuk, L., Clausen, J.H., Camtepe, S.A., Albayrak, S., Yildizli, C.: Smartphone Malware Evolution Revisited: Android Next Target? In: Proc. of MALWARE (2009)
46. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: Andromaly: a behavioral malware detection framework for android devices. Journal of Intell. Info. Syst. 37, 1–30 (2011)
47. Symantec. SymbOS.Cabir (June 2004), http://www.symantec.com/security_response/writeup.jsp?docid=2004-061419-4412-99
48. Tencent Mobile Security Lab. Disguised the explosive growth of the virus (October 2011) http://www.tastecate.com/freepages353623
49. TGDaily. Smartphone Malware at an All-Time High (December 2010), http://www.tgdaily.com/security-brief/53060-smartphone-malware-at-an-all-time-high
50. Vesselin: Dexid (2011), http://dl.dropbox.com/u/34034939/dexid.zip
51. VirusTotal, http://www.virustotal.com
52. Xie, L., Zhang, X., Chaugule, A., Jaeger, T., Zhu, S.: Designing System-Level Defenses against Cellphone Malware. In: Proc. of IEEE SRDS (2009)
53. Ye, S.: Android Market is Currently Blocked in China. Here are your Alternatives (September 2011), http://techrice.com/2011/10/09/android-market-is-currently-blocked-in-china-here-are-your-alternatives/

# A Static, Packer-Agnostic Filter to Detect Similar Malware Samples

Grégoire Jacob[1,3,4], Paolo Milani Comparetti[2,4], Matthias Neugschwandtner[2], Christopher Kruegel[1,4], and Giovanni Vigna[1,4]

[1] University of California, Santa Barbara
[2] Vienna University of Technology
[3] Télécom SudParis
[4] LastLine, Inc.
gregoire.jacob@gmail.com, {pmilani,mneug}@seclab.tuwien.ac.at,
{chris,vigna}@cs.ucsb.edu

**Abstract.** The steadily increasing number of malware variants is a significant problem, clogging the input queues of automated analysis tools. The generation of malware variants is made easy by automatic packers and polymorphic engines, which produce by encryption and compression a multitude of distinct versions. A great deal of time and resources could be saved by prioritizing samples to analyze, either, to avoid the repeated analyses of variants and focus on innovative malware, or, on the contrary, to re-analyze variants and have better insights on their evolution. Unfortunately, indexing in malware analysis tools and repositories relies on executable digests (hashes) that strongly differ for each variant.

In this paper, we present a robust filter to quickly determine when a malware program is similar to a previously-seen sample. Compared to previous work, our similarity measure does not require the costly task of preliminary unpacking, but instead, operates directly on packed code. Our approach exploits the fact that current packers use compression and weak encryption schemes that do not break, in the packed versions, all the similarities existing between the original versions of two programs. In addition, we introduce a packer detection technique that is able to distinguish between different levels of protection, such as unpacked, compressed, encrypted, and multi-layer encrypted code. This allows us to optimize the sensitivity of the similarity measure accordingly. We evaluated our approach on a large malware repository containing 795,000 samples. Our results show that the similarity measure is highly effective in filtering out malware variants, even after re-packing, and can reduce the number of samples that need to be analyzed by a factor of 3 to 5.

## 1 Introduction

Malware authors release an ever-increasing number of malware samples. Overwhelmed by the quantity (up to several thousands per day), malware analysts cannot rely on manual analysis to examine the characteristics and behavior of new malware samples. As a result, analysts use automated dynamic analysis

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 102–122, 2013.

tools such as *Anubis* [1], *CWSandbox* [2], *Norman Sandbox* [3], or *ThreatExpert* [4]. These tools monitor the execution of malware samples in a controlled environment and provide a detailed report of their activity (*e.g.*, interactions with processes, files, the registry, or the network). The drawback of the dynamic approach mainly lies in the execution time, especially considering that the instrumented environments used to confine malware are usually slower than "real" execution environments. A minimal execution time is hard to determine, but, usually, it takes several minutes before a malware sample performs enough suspicious operations to allow for a correct characterization of its behavior, plus the time necessary to revert the instrumented environment in a clean state.

Throwing more hardware at the problem offers only temporary relief, considering the growing number of variants produced by malware authors. Moreover, this approach is wasteful, as a majority of the released malware samples are simple variations of existing ones. It would be preferable to manage analysis priorities and spend the available resources, either on previously unseen malware, or, on similar variants to obtain insights on their evolution, such as finding new control servers for bots [19]. To this end, a technique is needed to quickly determine whether a submitted sample is similar to one that was analyzed before.

Different static approaches have been explored to address the problem of malware similarity. In [10], the authors introduce a distance-based approach that uses the edit distance between instruction sequences, whereas the approaches described in [13] and [26] rely on the cosine vector distance over $n$-gram distributions of instructions. Other approaches replace the one-to-one distance function with more complex classification algorithms [18,21,25]. In [7] and [11], the authors introduce a graph-based approach, which compares graph representations extracted from the disassembled code. Some of these systems are computationally expensive. More importantly, all these previous approaches require that the malicious code is *unpacked* and *disassembled* first. Unfortunately, existing generic unpackers rely on a dynamic instrumentation of executables [12,17,22], and thus also suffer from performance limitations due to the code execution.

In this paper, we present an efficient, static technique that can identify samples that are similar to those previously analyzed, without the need to execute them. That is, our similarity measure is directly computed over packed and encrypted samples. This is possible because existing packers and their compression/encryption algorithms retain some of the properties present in the original code. Thus, two packed executables, produced by a certain packer, are likely to remain similar (in certain ways) if they were originally similar. The work closest to our proposed filter is *peHash* [27], a system that also attempts to detect duplicate malware samples without executing them. To this end, *peHash* leverages the structural information extracted from malware samples, such as the number, size, permission settings, and Kolmogorov complexity of the sections in a PE executable.While this approach makes *peHash* efficient, its reliance on ephemeral features is not robust, and it can be trivially confused. Our similarity measure, on the other hand, is based on properties directly derived from the code (content) of the malware program, and hence, is more tamper resistant.

To summarize, our contributions are the following:

- We introduce an efficient and robust similarity measure for malware samples. The measure operates directly on the packed code section(s) of the program.
- We present a packer detection method that can also identify the type of algorithm: compression, encryption, multi-layer encryption. The detected type is used to automatically configure the sensitivity of the similarity measure.
- We discuss prefiltering methods to select samples candidate for comparison. Prefiltering relies on efficient heuristics to quickly discard irrelevant samples.
- We have evaluated our techniques over a large malware repository (795,000 samples). Our experiments demonstrate that our similarity measure is effective in filtering out packed variants obtained from the an original malware. The system reduces the number of samples to analyze by a factor of 3 to 5.

## 2 Similarity and Packing

Techniques to compute the static code similarity between malware samples face the same problems as static malware detection techniques: the packers and mutation engines that are widely used by malware writers to evade signature detection also blur the similarity between malware variants. According to [17], the percentage of malware that is packed has grown steadily, up to more than 80% of the samples currently found in the wild. Packers were first used to reduce the size of executables by compression. To hinder reverse engineering further, encryption was soon combined to compression. Encryption makes unpacking more difficult and, from the point of view of the malware authors, it increases the number of variants that can be generated by simply changing the encryption key. Recently, protections based on virtualization were introduced, where the original program is translated into virtual instructions that are then executed by an embedded virtual machine. In this work, we did not try to address virtualization-based packers, such as *Themida* or *VMProtect*, because they have complete control over the mapping between real and virtual instructions. In these conditions, code similarities at the binary level are hard to preserve.

In general, compression and encryption severely hinder any similarity computation on executables because the content of code sections is modified, both in terms of byte sequences and statistical properties. To better understand the ways in which current packers modify the body of an executable, we manually examined (reverse engineered) a number of popular tools frequently used by malware authors. A first, important observation is the limited number of algorithms that are at the core of current packers. For compression, dictionary-based approaches are the most widely used (mostly *LZ77*), sometimes combined with entropy and range encoders. For encryption, reversible arithmetic operations (such as *add/sub*, *rol/ror*, *xor* with 8-bits or 32-bits keys) are the most commonly used techniques. The use of stronger cryptographic algorithms (such as *RC4*, *DES*, or *AES*) is rare because these algorithms are much slower and need to be reimplemented to avoid using easily-detected cryptographic APIs.

**Table 1.** Impact of the different packing algorithms on the binary content

| | | |
|---|---|---|
| *Dictionary Compression* **e.g.** *LZ77* is used in *LZO*(*PolyEnE*), *NRV*(*UPX*). | Principle | Most-frequent bytes (or blocks) are replaced by relative references to previous occurrences. |
| | Alignment | Byte alignment is preserved by uncompressed blocks and references. |
| | Sequences | Order of incompressible blocks is preserved. Relative references are interleaved in between; their inter-space is bound by the size of the reference window. |
| | Distribution | Distribution is flattened because frequent blocks are replaced by references that tend to introduce infrequent byte values. |
| *Entropy Encoding* **e.g.** *Huffman* is combined with *LZ77* for *deflate*(*gzip*). | Principle | Most-frequent bytes (or blocks) are encoded by symbols (bit strings) of smaller size. |
| | Alignment | Byte alignment is destroyed by the shortened symbols. |
| | Sequences | Byte blocks are replaced by shorter symbols but their sequence is preserved. |
| | Distribution | Distribution is destroyed due to lost alignment, but can be reconstructed over the encoded symbols. |
| *Range Encoding* **e.g.** Encoding is combined with *LZ77* for *LZMA*(*NsPack*). | Principle | Most-frequent bytes (or blocks) are replaced by a single integer range representation. |
| | Alignment | Byte alignment is destroyed by the shortened integer ranges. |
| | Sequences | Sequences are shortened, but order remains. |
| | Distribution | Distribution is destroyed due to lost alignment, but can be reconstructed over the encoded ranges. |
| *Arithmetic Encryption* **e.g.** *PolyEnE* uses 32-bit $xor$, $add/sub$, $rot/rol$ for encryption. | Principle | Blocks of bytes are independently encrypted using reversible arithmetic operations. |
| | Alignment | Byte alignment is preserved by the key and blocks. |
| | Sequences | Sequences are preserved, except that blocks are replaced by their encrypted values. |
| | Distribution | $n$-gram distribution is permuted, where $n$ is the size of the encryption block. |
| *Key/Operation Variation* **e.g.** *Yoda's Cryptor* uses a cycle of $xor$, $add/sub$, $rot/rol$. | Principle | A different encryption key/operation is used for each new block. |
| | Alignment | Byte alignment is preserved by the key and blocks. |
| | Sequences | If variation is cyclic, repeated blocks at the same relative position in the cycle have the same encrypted value, otherwise, sequences are lost by the variable encryption. |
| | Distribution | If variation is cyclic, the effect is identical to encryption of larger blocks (encryption block length is equal to the cycle length). |
| *Multi-layer Encryption* **e.g.** *tElock* uses multiple layers of 8-bit $xor$ encryptions. | Principle | The entire input is encrypted multiple times. |
| | Alignment | Byte alignment is preserved by the key and the byte blocks. |
| | Sequences | If layers are aligned with same size of encryption blocks, effect is equivalent to a single encryption, otherwise, overlaps are equivalent to key variations. |
| | Distribution | If layers are aligned with same size of encryption blocks, effect is equivalent to a single encryption, otherwise, overlaps are equivalent to key variations. |

Table 1 presents the key algorithms used by packers, as well as their impact on the data contained in the sections of the program: 'alignment" indicates whether a byte-aligned data block remains aligned after packing, "sequences" discusses the effects of packing on the order of bytes (or regions) in the original program, "distribution" characterizes how the distribution of bytes (or $n$-grams) in the original binary is altered by the packing process. By combining compression and encryption, packers tend to destroy all similarity between an original executable and its packed version. However, looking closer at Table 1, we observe that some information is preserved both by compression and weak encryption algorithms.

**Compression Algorithms.** Dictionary-based packers preserve certain incompressible parts of the original program, while they compact other parts by replacing entire sub-sequences with references (relative offsets) to previously-seen, uncompressed occurrences of the same sub-sequence. If two executables are similar before packing, their incompressible parts will be mostly similar. As for the compressed parts, one can expect that most of the relative offsets point to similar positions. Dictionary-based algorithms align both uncompressed regions and offsets on byte boundaries. Entropy encoders operate by replacing frequent bytes (or blocks) with shorter symbols. The lengths of these symbols are typically not a multiple of a byte, and hence, the byte alignment is destroyed. This also significantly alters the byte distribution. However, when considering the encoded symbols, their distribution is identical to the byte (block) distribution of the data before encoding. Similar considerations hold for range encoders.

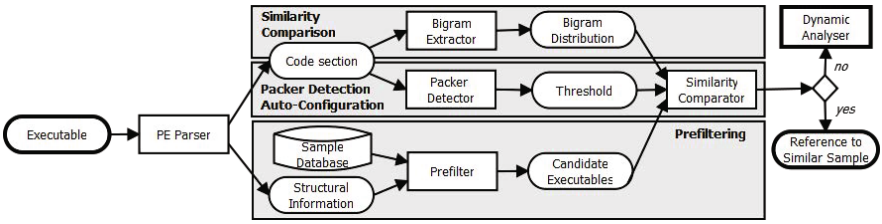



**Fig. 1.** Architecture of our similarity filter

**Encryption Algorithms.** The reversible arithmetic operations used by a majority of crypters only achieve a simple substitution of the byte blocks in the original code. Arithmetic encryption results in a permutation of the distribution of the original $n$-grams but the alignment of bytes is not affected. Most of these crypters do not implement any chaining to strengthen their algorithm. Based on the packers we have studied, only a few crypters were actually offering position-dependent encryptions: these crypters apply a short cyclical variation of the key or the arithmetic operation but no chaining.

The important conclusion that can be drawn from the previous observations is that packers preserve certain properties of the original code. Compressors tend to alter the byte alignment. However, when considering the compressed symbols, some sequences are incompressible by dictionary-based compression and references to compressed sequences are deterministically determined. Compressors thus preserve similarity because originally similar programs result in similar compressed data, and, consequently, similar symbol distributions. Crypters do not alter the byte alignment. However, they create a permutation of the distribution of bytes (blocks) in the original program, where the permutation depends on the encryption key. In the next section, we will discuss how we can leverage these insights to perform similarity computations directly on packed code.

## 3 A Similarity Measure for Packed Executables

The filter that we introduce in this paper is designed to detect malware programs that are similar to previously-seen samples. Leveraging this filter, we can prioritize submissions to dynamic analysis systems according to the samples novelty.

To compute the similarity between two (malware) programs, we compute the distance between their *code signals*. A code signal is a bigram distribution over the raw bytes of the code section, but extracted in a way to compensate for the modifications (noise) introduced by packers (Section 2). An overview of the system is presented in Fig. 1. We keep a database of previously-analyzed malware programs that stores, for each sample, its code signal. Whenever a new sample arrives, its code signal is extracted. We then compute the distance of this signal with respect to those stored within the database (Section 3.1). If this distance is below a certain threshold, a similar sample already exists in the database, and no further analysis is performed. If the distance is above the threshold, the sample is submitted for further analysis and the database is updated accordingly.

To increase the speed and precision of the system, two additional steps are introduced. First, we use a packer detector that automatically configures the distance sensitivity, based on the type of packing used (Section 3.2). Second, the distances are not computed for all samples in the database. Instead, a prefilter selects likely candidates to reduce the number of comparisons (Section 3.3).

### 3.1 Extracting and Comparing Code Signals

We employ *code signals* to characterize the executable section of a binary, and to determine the distance of binaries, from one to another. A code signal is a distribution of byte bigrams (pairs of subsequent bytes), extracted in a particular way from a program's code segment. One reason for operating directly on the raw bytes of the malware code is speed. Neither disassembly nor any other interpretation of the bytes is required. A second reason is that the similarity measure must be packer-agnostic, meaning that the measure should work directly on the packed code, which cannot be disassembled. To handle packed code, we introduce two specific transformations during the code signal extraction.

**Extracting Code Signals.** As discussed in Section 2, when similar programs are compressed or encrypted by current packers, the resulting binaries share certain similarities that "shine through" the packing process. We exploit these similarities using two transformations to respectively address the previously-identified problems of alignment destruction and distribution permutation.

*1) Bit-shifting window:* To recover from the destruction of the byte alignment, a bit-shifting window is used to extract bigrams, instead of the traditional byte-shifting window. The process is shown in Fig. 2. Using a byte-shift, any local difference between two similar streams of compressed data is likely to result in a disalignment because compressed symbols have sizes that are not byte-aligned. The importance of the bit-shift thus lies in its capacity to resynchronize two similar compressed streams with the correct alignment.

*2) Sorted distribution:* Once all bigrams are extracted from a malware's code section, we compute the bigram frequencies. Their distribution is then normalized by dividing these frequencies by the total number of bigrams in order to obtain a probability distribution. To address the possible, additional encryption of the code by simple arithmetic operations, the distribution is sorted by decreasing order of probability values. As mentioned in Section 2, for simple block encryption algorithms (without chaining), the $n$-gram distribution of the encrypted code is simply a permutation of the original distribution; in these cases, sorting the bigram distribution can perfectly recover the similarity between samples that was obscured by encryption. This technique was originally introduced in anomaly detection to detect similar attack payloads, possibly encrypted [15].

In our case, only a partial recovery of the distribution is possible because of the bit-shifting window used to extract bigrams: the bit-shifting is required to handle compression (and the alignment issues compression introduces). However, looking at the extracted bigram distributions, we find that only a small fraction of bigrams are frequent enough to contribute significantly to the code distribution

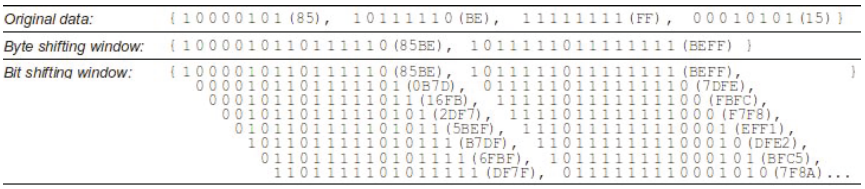



**Fig. 2.** Bigram extraction by bit shifting window

Let us consider a 4-byte value $X = X_1X_2$. $X$ is encrypted by a function $E$ as follows: $X' = E(X, K)$ with $K = K_1K_2$ and $X' = X'_1X'_2$.

$E$ **is xor:** Relation between encrypted values and inputs:
$X'_1 = X_1 \oplus K_1$ and $X'_2 = X_2 \oplus K_2$
No diffusion between bits of of $X_1$ and bits of $X_2$.

$E$ **is addition:** Relation between encrypted values and inputs:
$X'_1 = X_1 + K_1 + carry$ and $carry, X'_2 = X_2 + K_2$
If $carry = 0$, no diffusion between upper bits of $X_1$ and lower bits of $X_2$.
If $carry = 1$, only the rightmost bits of $X_1$ are impacted by the encryption of $X_2$.

$E$ **is rotation:** Rotation diffuses overflowing bits from one side to the opposite side.
Still, particular keys do not properly achieve diffusion:
If $K = \alpha 16$ and $\alpha$ is even then: $X'_1 = X_1$ and $X'_2 = X_2$
If $K = \alpha 16$ and $\alpha$ is odd then: $X'_1 = X_2$ and $X'_2 = X_1$

**Fig. 3.** Diffusion between upper and lower bytes for arithmetic encryption operations

(these are predominant bigrams). The remaining bigrams have a very small probability compared to these frequent (predominant) bigrams, and they are in the long tail of the distribution. As expected, the predominant bigrams are those bigrams that are aligned on instruction boundaries or on the boundaries of compressed/encrypted symbols. The bigrams with small frequencies typically correspond to bigrams that overlap adjacent instructions or symbols.

In our experiments, we found that only about 7% of the bigrams are frequent enough to contribute to the code distribution. If we restrict our view of the bigram distributions to these predominant bigrams, the sorting process is still efficient in recovering the significant part of the distribution. In cases where the size of the encryption blocks is equal to or smaller than the size of the bigrams, the predominant bigrams are simply permuted. If blocks are of a larger size than bigrams, the quality of the recovery for these predominant bigrams depends on the extent to which the encryption operation on a large block can be approximated as separate (independent) encryption operations on sub-blocks. When this approximation holds, the original probability of a bigram is divided between a limited number of encrypted bigrams, depending on the relative position of the bigram to the key. For example, the original probability of a bigram $X$, after encryption by xor with a 32-bit key $K = K_1K_2$, will be always divided between $X \oplus K_1$ and $X \oplus K_2$. The approximation in separate encryptions actually depends on the diffusion achieved by encryption between the bits of different sub-blocks. Fig. 3 discusses different conditions under which certain arithmetic operations do not achieve diffusion.

This technique is designed to address the simple encryption algorithms used by current packers. On the other hand, it does not the address standard encryption

algorithms, such as *AES* or *RSA*, used in contexts where the security of the data is critical and stronger cryptography is required.

**Comparing Code Signals.** The comparison between code signals is performed using Pearson's $\chi^2$ test:

$$\chi^2 = \sum_{i=0}^{2^{16}-1} \frac{(o_i - r_i)^2}{r_i} \; where \; r_i > 0 \tag{1}$$

where $o_i$ are elements of the distribution extracted from the submitted sample, and $r_i$ are elements of the reference distribution from the candidate samples. The cases where $r_i = 0$ were ignored since, as previously seen, they correspond to negligible bigrams that are not contributing significantly to the distribution.

We did consider a number of other similarity measures (such as the cosine vector distance), but we found that the $\chi^2$ test yielded the best results in terms of precision and performance. Moreover, we investigated weighting mechanisms, such as the *inverse document frequency*. Unfortunately, such mechanisms do not improve the results since compression and encryption make any *a priori* hypothesis about the statistical frequency of bigrams unreliable.

The $\chi^2$ measure is computed between the distributions of the submitted sample and the first candidate. If the test value remains below a given threshold $\tau$, the two samples are considered similar. Otherwise, the test is repeated with the next candidate. Whenever a similarity is found with one of the candidate samples, the comparison process is stopped, and the reference to the existing sample is returned. If no similarity is found, the comparison process is continued until the set of candidate samples is exhausted.

The actual value for the threshold $\tau$ is selected based on two factors. First, the threshold provides a mechanism to adjust the sensitivity of the filter, and hence, to control the trade-off between false negatives and false positives. In our use case, a false negative (failing to recognize that a similar sample is already in the database) is much less problematic than a false positive (incorrectly concluding that a similar sample is already in the database). This is because, in case of a false negative, a duplicate sample is analyzed, which results in a small waste of resources. In case of a false positive, a new, and possibly interesting sample, is incorrectly discarded. The second factor is the output of the packer detector (discussed in the next section). We use a set of different thresholds that are optimized according to the packing level of the tested program.

### 3.2 Packer Detection

As explained in Section 2, packers modify the byte distribution of the code. In particular, packing often leads to a "flatter" distribution. In case of compression, frequent values are replaced by references or short symbols. In case of encryption, the same, frequent byte value might be mapped to different, encrypted values. Flatter distributions can lead to false positives, because the similarity values returned by the $\chi^2$ test decrease (compared to unpacked samples). The similarity threshold should thus be reduced accordingly when checking packed executables.

$T_1$: *Uncertainty test.* Code entropy.
$T_2$: *Uniformity test.* $\chi^2$ test between the code and an equiprobable distribution.
$T_3$: *Run test.* Longest sequence of identical bytes in the code.
$T_4$: $1^{st}$*-order dependency test.* Autocorrelation coefficient of the code at lag 1.

| Type | Test series | Detection criterion |
|---|---|---|
| Packers | $T_1 : H \geq t_1$ | `packed` if $T_1$ = true |
| Compressors | $T_2 : U < t_2$<br>$T_3 : lgth(run) \leq t_3$<br>$T_4 : \lvert ACC \rvert < t_{4a}$ | `compressed` if `packed` $\wedge$<br>no more than one of $T_2$,$T_3$,$T_4$ = true |
| Crypters | $T_2 : U < t_2$<br>$T_3 : lgth(run) \leq t_3$<br>$T_4 : \lvert ACC \rvert < t_{4a}$ | `encrypted` if `packed` $\wedge$<br>two or more of $T_2$,$T_3$,$T_4$ = true |
| Multi-layer crypters | $T_4 : \lvert ACC \rvert < t_{4b}$ | `multi-layer` if `encrypted` $\wedge$ $T_4$ = true |

**Fig. 4.** Statistical tests for packer detection

To detect packed executables, we leverage the fact that a flattened distribution makes packed code similar to random data. Thus, the statistical properties used to assess random generators can be used to detect packed executables, and classify their type of protection: compression, single and multi-layer encryption.

**Packer Detection and Classification.** To detect packers and to identify the type of protection, we introduce four statistical tests in Fig. 4. These tests are performed over the raw bytes in the actual code section or the packed code section (depending on whether the sample is packed).

The entropy-based test $T_1$ is the traditional test used to detect packed executables. A high entropy value constitutes a significant sign of randomness. Thus, whenever $T_1$ yields a code entropy value above an experimentally determined threshold, the sample is considered packed. For all packed samples, we use three additional tests $T_2, T_3$, and $T_4$ to determine more precisely the type of packing. These tests were originally designed for assessing random number generators [23]. Here, we apply them in a novel context.

The uniformity-based test $T_2$ and the run-based test $T_3$ are primarily employed to distinguish between compressed and encrypted code. When an encryption algorithm uses input blocks that span multiple bytes, one particular (byte) value in the original code is likely mapped to several different, encrypted values in the packed code, depending on the relative positions of the bytes in the encrypted block. Thus, the distribution of encrypted code is closer to a uniform distribution than compressed code (a larger specter of observed bigrams with a levelled frequency), a property checked by $T_2$. Moreover, some compression algorithms (especially dictionary-based approaches) can produce sequences (runs) of identical bytes, something that is unlikely for crypters. As a result, the presence of longer runs of identical bytes is an indication of compression.

Finally, executable code is known to have a first-order dependency [21]. This dependency between consecutive bytes is partially destroyed by compression and encryption. In the case of multi-layer crypters, the boundaries between different layers introduce additional discontinuities. These discontinuities are detected by testing the autocorrelation coefficient (ACC) of the code $T_4$. Fig. 4 explains how

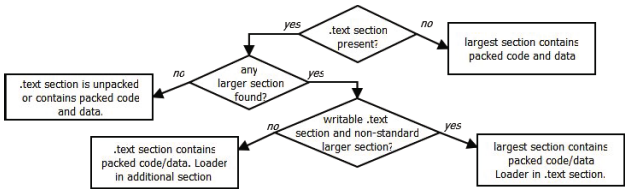



**Fig. 5.** Finding the plain/packed code section

our four tests are combined to identify the level of packing. Thresholds $t_1$, $t_2$, $t_3$, $t_{4a}$, $t_{4b}$ are experimentally determined in the evaluation section.

**Locating Packed Code.** The four statistical tests have to be performed on the "normal" text (code) segment for unprotected executables or on the section that holds the packed code. Since packers modify the sections of executables, the risk is to perform the test over the section that contains the loader. Fig. 5 shows our heuristic to find the section that contains the packed data. Notice that the identified section is later used to extract the code signal.

### 3.3 Fast Prefilter

Computing the similarity of a new malware sample with respect to those already stored in the database is potentially costly when the number of samples increases. To reduce the necessary similarity computations, but also to reduce potential false positives due to random collisions between code signals, we apply a prefilter to select only a subset of candidate samples for further consideration. This prefilter step uses fast heuristics to discard non-similar samples based on straightforward observations. More precisely, the prefilter uses two sequential heuristics: a first heuristic based on the size of samples, and a second heuristic based on the structural information contained within the programs' PE headers.

**Size-Based Filtering.** An immediate criterion of similarity between PE executables is their size. When malware writers produce variants of their original code, these variants tend to be of similar size. Of course, the size of samples derived from the same original source code might change because of compilation parameters, small modifications to the code, and, most importantly, because of packing. Taking into account these factors, we compute, for a new sample, a range with limits that are a fixed percentage above and below this sample's size. The prefilter then selects candidate samples whose size falls within this range. If no candidate is found, the sample is considered new.

**PE-Characteristic-Based Filtering.** Further criteria of similarity between executables are their structural characteristics. In the PE format, the header contains important information about the executable's layout, both on disk and in memory, and meta-information about the compilation process. However, only a subset of these features is useful for prefiltering: we only consider features that provide sufficient differentiation between executables while being robust to packing (that is, features that are not modified by packers). The 16 features we

selected are presented later in Table 5. The prefilter computes the Hamming distance between the PE features of a new (incoming) sample and the features of all candidates that were selected by the first heuristic. When the distance is larger than a threshold, the corresponding database sample is discarded. All remaining samples become candidates for the similarity measure computation.

## 4 Evaluation

The filter presented in previous sections was implemented and used to process samples submitted to an automated, dynamic malware analysis system. The evaluation was carried out in two steps. For the first step, we used our filter on known samples for which ground truth was available (Section 4.1). The goal of this first step was to establish the similarity thresholds and configure the packer detector as well as the prefiltrer. For the second step, we applied the filter to a large collection of malware samples that were provided to us by the authors of *Anubis* [1] (Section 4.2). The goal of this second step was to verify that the precision is maintained in real-world conditions, when the filter is exposed to a large number of diverse malware samples and packers. We also took advantage of this second step to study the scalability and the robustness of our approach.

### 4.1 Experiments on Known Samples

We started our experiments with two data sets. The first set, $S_1$, contained 384 PE executables, mostly taken from the system directory of a *Windows XP* installation. It also contained open-source software, such as *OpenOffice*, and free shareware, such as *mIRC*. All programs in $S_1$ were unpacked and served as examples of dissimilar (unrelated) binaries.

The second set, $S_2$, contained 65 bots, whose source code was made available to us. These bots belong to two malware families: *SdBot* (23 samples) and *rBot* (42 samples). The *SdBot* samples were further classified as versions 4 and 5, while the *rBot* samples span five versions ranging from 3 to 7. Since the samples in $S_2$ are related to various degrees, we could leverage this data set as labeled ground truth to study the precision of our similarity measure. Any other malware family with a version history could have been used for this configuration.

**Packer Detection.** To assess our packer detection technique, we selected seven packers, based on their popularity with malware writers: *UPX*, *FSG*, *NsPack*, *WinUPack*, *Yoda's Cryptor*, *PolyEnE* and *tElock*. We also added instances of the *Allaple* worm as a representative example for polymorphic malware; its engine uses techniques similar to packing. Table 2 provides an overview of these packers, covering the compression and encryption algorithms they implement: four compressors, two crypters, and two multi-layer crypters. Looking at prevalences, these eight packers cover 86% of the packed samples from the *Anubis* data set.

We first packed each of the 384 executables from $S_1$ with the 7 packers, and then added 120 *Allaple* samples. This set of packed executables was used to verify the rate of False Negatives (FN) of the technique. The unpacked versions of these executables were then included to the data set in order to verify the rate of False

**Table 2.** Specifications of the tested packers

| Name | Algorithms | Sections | IAT | Preval. |
|---|---|---|---|---|
| UPX | -compression (NRV(LZ77)) | -loader section ".UPX1"<br>-compressed code/data".UPX1" | API call redirection | 38.41% |
| FSG 2.0 | -compression (aPlib(LZ77)) | -loader section ""<br>-compressed code/data "" | API call redirection | 14.00% |
| NsPack | -compression (LZMA(LZ77 + range encoding)) | -loader section ".nsp0"<br>-compressed code/data".nsp1" | API call redirection | 1.89% |
| WinUpack | -compression (LZMA(LZ77 + range encoding)) | -loader random<br>-compressed code/data random | Api call redirection | 2.09% |
| Yoda's Cryptor 1.3 | -combined encryption (add, xor, rol)<br>-compression only in yProtector | -sections maintained<br>-loader section "yC" | API call redirection | <1% |
| PolyEnE | -compression (LZ77)<br>-random encryption (add, xor, rol) | -sections maintained<br>-loader section ".Polyene" | API call redirection | <1% |
| tElock | -compression (aPLib(LZ77))<br>-multi-layer encryption (add, xor, rol) | -sections maintained<br>-loader section "" | API call redirection | <1% |
| Allaple | -compression (aPLib(LZ77))<br>-multi-layer encryption (xor) | -loader section ".text"<br>-compressed code/data ".data" | API call redirection | 30.22% |

**Table 3.** Detection/classification of packers

| Name | Unpacked | Packed | Compr. | Crypt. | MLCrypt. |
|---|---|---|---|---|---|
| Unpacked | 99.74% | 00.26% | 00.26% | 00.00% | 00.00% |
| FSG | 18.18% | 81.82% | 81.55% | 00.00% | 00.27% |
| UPX | 03.04% | 96.94% | 96.10% | 00.56% | 00.28% |
| NsPack | 12.11% | 87.89% | 87.63% | 00.26% | 00.00% |
| WinUPack | 13.84% | 86.16% | 83.55% | 02.09% | 00.52% |
| Compressors | 11.80% | 88.20% | 87.21% | 00.72% | 00.27% |
| YodaCryptor | 17.99% | 82.01% | 06.08% | 74.87% | 01.06% |
| PolyEne | 06.01% | 93.99% | 28.98% | 62.14% | 02.87% |
| Crypters | 12.00% | 88.00% | 17.53% | 68.51% | 01.96% |
| tElock | 04.84% | 95.16% | 00.57% | 70.94% | 23.65% |
| Allaple | 00.00% | 100.0% | 00.00% | 72.22% | 27.78% |
| Multi-layers | 02.42% | 97.58% | 00.28% | 71.58% | 25.72% |
| Packed | 08.74% | 91.26% | N/A | N/A | N/A |

Positives (FP). Training over a small subset of this data set, we obtained the following thresholds that optimize the trade-off between FN and FP: $t_1 = 4.73$, $t_2 = 0.0012$, $t_3 = 2$, $t_{4a} = 0.005$, $t_{4b} = 0.002$ *c.f.* Fig. 4, Section 3.2.

The detection results for the remaining samples (test set) are presented as a confusion matrix in Table 3. One can see that the detector is able to distinguish very well between unpacked and packed executables: the detection rate for unpacked samples is 99.74%, while it is over 91% on average for packed programs. Furthermore, our statistical tests were able to correctly distinguish, in more than 80% of the cases, between compressors and crypters. The lowest classification rate was achieved for multi-layer crypters. The reason is that encrypting the same executable multiple times does not necessarily result in stronger encryption. In particular, several layers of *xor* encryption are basically equivalent to a single layer. It is important to observe though that a misclassification only leads to the use of a suboptimal threshold, but it does not prevent the system from computing correct similarity results.

When putting our detection results into the context of related work, our technique provides fine-grained distinctions between different types of packing without making use of packer-specific signatures or features. Systems such as [16]

relying on pure entropy, or [20] relying on structural properties of the executable, only distinguish between packed and regular code. More advanced systems such as [8] or [24] can precisely classify packers by name using randomness profiles. However, these systems need to be trained for each individual packer that should be recognized, something that our system, providing a coarser-grained distinction, does not require because it relies on information theoretic metrics that extend to any other packer that uses similar algorithms.

**Tuning the Filter Granularity.** The goal of the next experiment is to select suitable filter thresholds. For this, we turned our attention to $S_2$, the set of 65 classified bot samples. More precisely, to build our configuration set, these 65 bots, together with all benign 384 programs from $S_1$, were packed with all seven packers and submitted to the filter.

$$TH = \frac{\textit{nb similar samples flagged as similar} + \textit{nb unique samples flagged as unique}}{\textit{nb submitted samples}}$$

$$FH = \frac{\textit{nb dissimilar samples flagged as similar}}{\textit{nb submitted samples}}$$

$$M = \frac{\textit{nb similar samples flagged as dissimilar}}{\textit{nb submitted samples}}$$

Granularity levels:
- $(f)-$ *two samples are similar if they belong to the same family*
- $(v)-$ *two samples are similar if they belong to the same family and have the same version*

To measure the filter precision, we use the following metrics: (i) the rate of True Hits, $TH$, which correspond to the cases where the filter successfully discards similar samples, or forwards new, unique samples to the analysis tool; (ii) the rate of False Hits (or false positives), $FH$, which correspond to the cases where new samples are discarded even though they are novel (these errors are critical, because they may result in a loss of interesting information); (iii) the rate of Misses (or false negatives), $M$, which correspond to cases where samples are forwarded to further analysis even though they should have been discarded (these errors are less severe, because they only result in unnecessary analyses).

In Table 4, we present the results of our experiments for two different sets of thresholds. The first set of thresholds corresponds to what we refer to as *family granularity*. That is, the thresholds are set with the aim of recognizing as similar two samples when they belong to the same malware family. That is, a sample that belongs to *rBot* version 5.0 should be considered similar to an *rBot* version 6.0. The thresholds were found by an optimization process that maximizes the rate of true hits while maintaining the rate of false hits under 0.5%. The rate of false hits is the most critical factor because it eventually corresponds to the potential loss of information we tolerate by not running a unique sample.

With family granularity, we observe 95.2% of true hits on average, with only 4.5% misses and, more importantly, only 0.3% of false hits. The rate of true hits indicates to which extent similarity is preserved by packers, even after the minor modifications brought to the code of the different versions. Unsurprisingly, the best results are observed for compressors, because their packing process is deterministic. On the other hand, the filter does not achieve 100% detection in the case of crypters because the size of the encryption key is typically 32-bits, which

**Table 4.** Precision of the similarity measure for various packers

| | Family granularity thresholds | | | | | | | Version granularity thresholds | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Packer | Thrsh. | TH(f) | FH(f) | M(f) | TH(v) | FH(v) | M(v) | Thrsh. | TH(v) | FH(v) | M(v) |
| None | 0.0020 | 99.8% | 00.2% | 00.0% | 94.2% | 05.8% | 00.0% | 0.0012 | 98.0% | 00.2% | 01.8% |
| FSG | 0.0018 | 99.6% | 00.4% | 00.0% | 91.5% | 08.5% | 00.0% | 0.0008 | 94.2% | 00.4% | 05.4% |
| UPX | 0.0018 | 91.8% | 00.2% | 08.0% | 89.9% | 02.1% | 08.0% | 0.0008 | 91.1% | 00.4% | 08.5% |
| NsPack | 0.0018 | 99.4% | 00.2% | 00.4% | 93.6% | 06.0% | 00.4% | 0.0008 | 94.7% | 00.2% | 05.1% |
| WinUPack | 0.0018 | 99.2% | 00.4% | 00.4% | 93.6% | 06.0% | 00.4% | 0.0008 | 94.7% | 00.2% | 05.1% |
| YodaCryptor | 0.0015 | 89.3% | 00.0% | 10.7% | 90.4% | 00.2% | 09.4% | 0.0006 | 90.2% | 00.0% | 09.8% |
| PolyEne | 0.0015 | 90.0% | 00.4% | 09.6% | 90.6% | 01.2% | 08.2% | 0.0006 | 89.8% | 00.4% | 09.8% |
| tElock | 0.0013 | 96.1% | 00.6% | 03.3% | 95.1% | 02.9% | 02.0% | 0.0004 | 91.8% | 00.2% | 08.0% |
| Allaple | 0.0013 | 92.2% | 00.0% | 07.8% | 82.2% | 10.0% | 07.8% | 0.0004 | 76.6% | 00.0% | 23.4% |
| Average | - | 95.2% | 00.3% | 04.5% | 91.3% | 04.7% | 04.0% | - | 91.3% | 00.2% | 08.5% |

**Table 5.** PE Header characteristics selected for comparison

| Location | Name | $H$ | $card$ | Name | $H$ | $card$ |
|---|---|---|---|---|---|---|
| *DOS Header* | `AddressNewExeHeader` | 1.87 | 13 | | | |
| *NT Header* | `Characteristics` | 0.67 | 7 | | | |
| *Optional Header* | `(min/maj)LinkerVersion` | 0.68 | 6 | `CodeBase` | 0.93 | 6 |
| | `ImageBase` | 0.44 | 5 | `(min/maj)OSVersion` | 0.43 | 4 |
| | `(min/maj)ImageVersion` | 0.46 | 4 | `(min/maj)SubsystemVersion` | 0.45 | 4 |
| | `Subsystem` | 0.22 | 2 | `DllCharacteristics` | 0.75 | 7 |
| | `SizeStackReserve` | 0.31 | 4 | `SizeStackCommit` | 0.44 | 5 |

is twice the bigram size. In this case, as we have seen, the similarity preservation depends on some bias of the encryption algorithm. The worst results are obtained for Yoda's Cryptor, because this crypter uses a cycle of different encryption operations and keys per block. The cycling operations make encryption position-dependent, thus explaining the higher rate of misses.

Depending on the desired level of granularity, it might be preferable to analyze different versions of the same malware family. In this case, the family granularity thresholds are too loose. This can be seen by looking at the false hit rates for malware versions, denoted as $FH(v)$, which reaches 4.7% when using family granularity thresholds. To differentiate between different malware versions, we created a second set of tighter thresholds (referred to as *version granularity*). It can be seen that, using these thresholds, $FH(v)$ drops to 0.2%. However, we also have to accept that the rate of misses increases. Notice that misses remain tolerable because they only imply re-running an existing sample, without potential loss of interesting information.

We also examined the precision of our system when analyzing the polymorphic worm *Allaple*, which was a major issue in 2007-2008, polluting malware repositories with thousands of mutated variants. The experiments have been run over two versions, namely *Allaple.b* and *Allaple.e*. The results are also given in Table 4. The worm variants are accurately detected in more than 92% of the cases, with a good distinction between versions.

**Configuring the Prefilter.** The size range that constitutes the first heuristic of the prefilter was configured so that the variants of a given program fall within this range, while it remains tight enough so that the number of irrelevant candidates

remain minimal. Considering the packed versions of the bot variants from $S_2$, the maximum size variation that we observed was 4.4%, which corresponds to a lower bound of 95.6% and a higher bound of 104.4% of the original size.

To find the best structural features to constitute the second heuristic of the prefilter, we again examined the original and packed versions of the samples from $S_1$. For all PE header fields, we verified that they were resilient to packing, and that they were distinguishing enough (sufficient number of different values, *card*, and high entropy, $H$). Table 5 provides the list of 16 selected features that are compared by Hamming distance with a threshold of 0.

### 4.2 Large Scale Experiments

The experiments with known samples allowed us to analyze the accuracy of our filter, tune detection thresholds, and configure the prefilter. In the next step, we performed a large-scale experiment with 794,665 malware samples that were submitted to the *Anubis* analysis tool in 2009. For each of these samples, we had at our disposal behavioral information (execution traces) and a reference clustering [6]. This clustering partitioned the malware programs into 91,522 different groups sharing similar runtime activity.

**Precision and Scalability.** We applied our filter to the entire data set of almost 795 thousand malware samples. To evaluate the filter precision, we use the aforementioned metrics: True Hits ($TH$), False Hits ($FH$), and Misses ($M$).

A problem for this experiment was the fact that we did not have ground truth available (such as source code or reliable malware labels). To address this, we introduced a reference classification based on the behavioral and structural information of executables. More precisely, we leveraged the behavioral clusters [6]: we considered two samples as similar when they produced similar behaviors, and hence, ended up in the same behavioral cluster. The behavior similarity was computed using the Jaccard distance between their execution traces. Unfortunately, the execution of malware programs is not deterministic and can change depending on the environment, time, or the availability of network resources (such as C&C servers). As a result, similar samples might end up in different behavioral clusters. Thus, to improve the reference clustering, we also considered structural characteristics of the malware programs. More precisely, we checked whether the executable sections of two programs share the same name, size, position in memory, and hash of the sections' contents. We considered two samples as similar when at least 90% of their structural information is identical and they share more than 70% of their behavior.

*Precision.* Table 6 shows the filter precision for three sets of thresholds. The first two correspond to the thresholds for *family* and *version* granularities, respectively, while the third is an extra set with more conservative thresholds. These three sets represent different trade-offs between reducing unnecessary analysis runs ($TH$) and the risk of discarding potentially interesting samples ($FH$).

For the first thresholds, the filter achieves a true hit rate of more than 90%. That is, more than 90% of similar (irrelevant) samples are correctly discarded.

**Table 6.** Filter accuracy for selected thresholds

(U-Unpacked, C-Compressed, E-Encrypted, MLE-Multi-Layer Enc.)

| Similarity Thresholds | | | | Family accuracy | | Version accuracy | | Misses | Reduction |
|---|---|---|---|---|---|---|---|---|---|
| U | C | E | MLE | TH(f) | FH(f) | TH(u) | FH(u) | M | Factor |
| 0.0020 | 0.0018 | 0.0015 | 0.00130 | 91.1% | 00.7% | 89.8% | 02.0% | 09.2% | 4.84 |
| 0.0012 | 0.0008 | 0.0006 | 0.00040 | 84.6% | 00.5% | 83.8% | 01.3% | 14.9% | 3.79 |
| 0.0005 | 0.0003 | 0.0002 | 0.00008 | 74.4% | 00.3% | 74.0% | 00.7% | 25.3% | 2.71 |

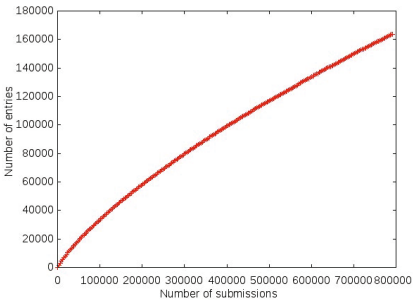



**Fig. 6.** Database growth

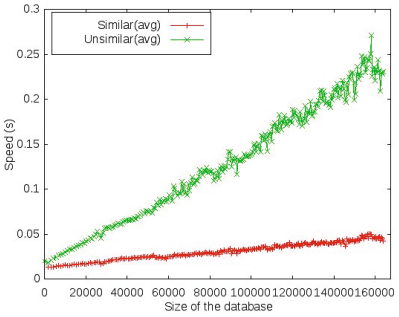



**Fig. 7.** Time/Submission

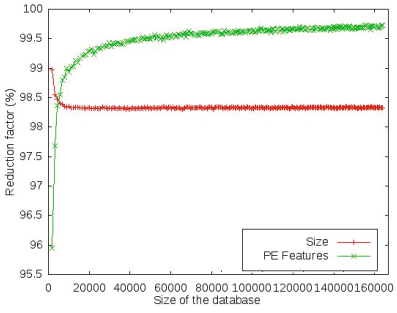



**Fig. 8.** Prefilter reduction

This leads to a reduction of the amount of overall analysis runs by a factor of almost five – saving a significant amount of valuable resources. This is paid for by a false hit rate of 0.7%. When the thresholds are more conservative, the number of incorrectly discarded samples ($FH$) is reduced to 0.3%. This, however, also lowers the hit rate, and thus, the reduction factor that can be achieved.

We then analyzed the False Hits produced by our filter in more detail. We found that incorrect similarities can be explained either by the failure of the heuristic to find the section containing the packed code (∼10% of FH), or, in most cases, by the misclassification of samples that, although they belong to different families, are part of the same class of malware (∼90% of FH). The heuristic failed mainly on very small executables where the packed code was negligible compared to the loader code. The misclassification mainly happened for fake anti-virus

software and IRC bots, probably because they share substantial portions of code. For this analysis, we used the malware labels produced by more than 40 AV scanners (run by *VirusTotal* [5]). We declared a false hit every time less than 5 scanners would agree on the family name.

With respect to misses, we found that most cases were caused by similar samples that exhibit similar dynamic behavior but were protected by different packers. For a given executable, the filter tends to create a new database entry for each different packer (type) used to protect this binary.

*Scalability.* To understand the scalability of our approach, we first examined the growth of the sample database. According to Figure 6, the database size increases sub-linearly with the number of submissions. Figure 7 shows a linear increase of the computation time with the number of entries. The computation time for similar samples is lower because, as soon as a similar entry is found, the computation stops. In the worst case, for unique samples, the filter takes no more than 300ms. This is 1,200 times faster than the 6 minutes required to execute a sample within *Anubis*. Considering the observed slowdown in the increase of the database size, the system should scale at least to tens of millions of samples.

The prefilter plays an important role in these performances. In Figure 8, it can be seen that the two heuristics reduce the candidate set to less than 1% of the database samples. Moreover, the figure shows that the prefilter maintains its effectiveness independently of the size of the database.

**Robustness of the Filter.** In the next step, we compare our approach to existing techniques that aim to detect similarities between malware binaries without analyzing their runtime behavior. To this end, we reimplemented *peHash* [27]. This tool operates mostly on structural characteristics of malware samples, and hence, does not require unpacking or disassembling the code beforehand. To understand how much the precision of our filter suffers because it has to operate on packed code (bytes) instead of disassembled instructions, we implemented a second version of our filter, where the bigram distribution (code signal) is not computed over the raw (and possibly packed) bytes, but over bigrams of disassembled instructions. This technique is similar to *Vilo* [26]. Finally, to compare with an alternative approach to detect malware similarity, we used an existing tool that operates on control flow graphs [14].

In the following, we refer to the four systems under examination as: *Filter* for our tool, *peHash*, *Disasm* for the disassembled version of *Filter*, and *Graph*. To experiment with these systems, we selected a subset of 18,645 samples from our real-world data set, where the corresponding unpacked binaries were available, as a byproduct of their execution in a dynamic analysis environment.

*Attacker model.* Here, we assume an attacker who develops a packer that operates directly on executables. That is, the packer is given a binary, and it has to output variants that cannot be recognized as similar. This is realistic because malware authors typically distribute their malware programs as binaries, using third-party packers to produce new variants on the fly.

**Table 7.** Compared robustness to structural modifications

| Modifications | *peHash* | *Filter* |
|---|---|---|
| Section permissions | 7.8% | 99.8% |
| Size of sections | 42.5% | 98.4% |
| Random data | 37.8% | 80.8% |
| Appended sections | 0.0% | 84.6% |

**Table 8.** Compared precision and runtime

| Systems | TH | FH | M | Time |
|---|---|---|---|---|
| No prerequisite on the code | | | | |
| Distance-based(*Filter*) | 80.8% | 00.7% | 18.5% | 6 min |
| Hash-based(*peHash*) | 81.1% | 00.6% | 18.3% | 9 min |
| Unpacked and disassembled code (* without unpacking) | | | | |
| Distance-based(*Disasm*) | 84.3% | 00.5% | 15.2% | 239 min* |
| Graph-based(*Graph*) | 83.4% | 00.4% | 16.2% | 847 min* |

**Table 9.** Compared robustness summary

| Modifications | *Disasm* | *Graph* | *peHash* | *Filter* |
|---|---|---|---|---|
| Modifying section permissions | ✓ | ✓ | × | ✓ |
| Changing section sizes | ✓ | ✓ | × | ✓ |
| Injecting data in sections | ✓ | ✓ | × | * |
| Appending new sections | ✓ | ✓ | × | * |
| Compression | × | × | ✓ | ✓ |
| Arithmetic encryption | × | × | ✓ | ✓ |
| Chained encryption | × | × | × | × |
| Strong encryption | × | × | × | × |

The need to operate on binaries imposes certain constraints; in particular, *the memory layout of the executable must be preserved*. Otherwise, addresses in the code or data sections would not resolve properly, and the program would crash. To work around this problem, the attacker would have to perfectly disassemble any input binary, which is extremely difficult in practice. Given this limitation, the attacker can perform structure-based operations leaving the original code untouched (1−4), and content-based operations that modify the code (5):

(1) Modifying access permissions of sections.
(2) Changing the size of sections on disk only.
(3) Injecting random data within the padding spaces.
(4) Appending sections at the end of the memory image.
(5) Compressing and/or encrypting code/data sections.

*Structure-based robustness.* To examine the techniques robustness to structure-based modifications, we developed an obfuscation tool that can apply all four structural modifications defined within our attacker model. Using this tool, we generated four kinds of variants for the 18,645 samples in our test set, and submitted them to *peHash* and our *Filter*. We did not test *Disasm* and *Graph* against the obfuscated binaries since these tools ignore structural information.

Table 7 presents the percentages of similar variants correctly identified for each type of modification. Overall, our approach is significantly more robust than *peHash*. This is not surprising since *peHash* focuses on structural information, which is easy to tamper with. Our *Filter*, on the other hand, relies on the statistical properties of the code, which are harder to change.

Table 7 also shows that our system considered as different a number of samples that should have been recognized as similar. The first, and main, reason was that the sizes of the binaries were changed by the obfuscator so that they exceeded the size range of the prefilter. To handle this issue, we can increase the size

range that the prefilter accepts, at the expense of a small performance penalty. The second reason, far less frequent, was that our heuristic to identify the packed code section (see Figure 5) was misled.

*Content-based robustness.* All packers apply some form of content-based obfuscation, by compression or some simple form of encryption. Since *Disasm* and *Graph* work only on unpacked samples, such simple transformations would already be sufficient to render them useless. However, in this section, we explore the precision of these systems when operating on unpacked binaries, compared to our *Filter* and *peHash* that operate on the corresponding packed versions.

For this experiment, we submitted the packed and unpacked versions of our 18,645 samples to all four systems. Table 8 compares the results, both in terms of precision and runtime. *PeHash* performs quite similarly to our approach, but at the significant expense of structural robustness, as was discussed previously. *Disasm* and *Graph*, which operate on unpacked executables, do not achieve a significantly better accuracy; in fact, the overall differences are minimal. This is encouraging because it shows that our *Filter*, working on packed code, produces almost the same results as tools that require unpacking and disassembling the malicious code. Moreover, the runtime of these tools is an order of magnitude larger, even when the unpacking time is not included.

These satisfying results are mainly explained by the fact that packers still rely on weak encryption algorithms. These results may no longer hold if packers start using stronger encryption algorithms such as *AES* or *RSA*, or, at least, design more clever algorithms such as in the case of blending attacks [9]. Blending attacks manipulate content, starting from an initial attack payload, until the payload satisfies a given distribution. In our case, blending attacks could be used to craft similar malware code distributions, making the filter ineffective. In their paper, the authors suggest the possibility of crafting the distribution by substitution operations and padding. In our case, the padding is however limited by the boundaries of the binary sections. To conclude this discussion about the filter robustness, Table 9 provides a summary view that compares the robustness of the four different systems that we examined with respect to our attacker model: ✓ if the system is robust, ✗ otherwise. The stars (*) in the table correspond to modifications to which the system is not entirely robust.

## 5 Conclusion

In this paper, we introduced an accurate, robust, and efficient technique for detecting similarity between malware samples. We leverage the fact that current malware packers only employ compression and weak encryption, and, therefore, information about the original program can be extracted from a packed binary. Unlike previous work [7,11,13,26], our technique is thus able to directly operate on packed binaries, avoiding the costly unpacking process. By doing this, our system is able to filter submissions to malware repositories or automated dynamic analysis tools. Large-scale experiments with almost 795,000 malware samples demonstrate that the filter achieves a significant reduction of the samples that need to be analyzed, with only a small amount of false positives.

## References


1. ANUBIS, http://anubis.iseclab.org
2. CWSandbox, http://www.mwanalysis.org
3. Norman Sandbox, http://www.norman.com/technology/norman_sandbox/
4. ThreatExpert, http://www.threatexpert.com
5. VirusTotal, http://www.virustotal.com
6. Bayer, U., Comparetti, P., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Proc. Symp. Network and Distributed System Security, NDSS (2009)
7. Carrera, E., Erdelyi, G.: Digital genome mapping. In: Virus Bulletin (2004)
8. Ebringer, T., Sun, L., Boztas, S.: A fast randomness test that preserves local detail. In: Virus Bulletin (2008)
9. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: USENIX Security Symposium (2006)
10. Gheorghescu, M.: An automated virus classification system. In: Virus Bulletin (2005)
11. Hu, X., Chiueh, T., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proc. ACM Conf. Computer and Communications Security, CCS, pp. 611–620. ACM (2009)
12. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: Proc. ACM Workshop Recurring Malcode, WORM, pp. 46–53. ACM (2007)
13. Karnik, A., Goswami, S., Guha, R.: Detecting obfuscated viruses using cosine similarity analysis. In: Proc. Asia Int. Conf. Modelling & Simulation, AMS, pp. 165–170. IEEE Computer Society (2007)
14. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006)
15. Kruegel, C., Vigna, G.: Anomaly detection of web-based attacks. In: Proc. ACM Conf. Computer and Communications Security, CCS. ACM (2003)
16. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Security and Privacy 5(2), 40–45 (2007)
17. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Proc. Annual Computer Security Applications Conf., ACSAC, pp. 431–441 (2007)
18. Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Japkowicz, N., Elovici, Y.: Unknown malcode detection and the imbalance problem. J. Computer Virology 5(4), 295–308 (2009)
19. Neugschwandtner, M., Comparetti, P.M., Jacob, G., Kruegel, C.: FORECAST – Skimming off the malware cream. In: Proc. Annual Computer Security Applications Conf., ACSAC (2011)
20. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recognition Letters 29(14), 1941–1946 (2008)
21. Krishna Sandeep Reddy, D., Dash, S.K., Pujari, A.K.: New Malicious Code Detection Using Variable Length $n$-grams. In: Bagchi, A., Atluri, V. (eds.) ICISS 2006. LNCS, vol. 4332, pp. 276–288. Springer, Heidelberg (2006)
22. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: Annual Computer Security Applications Conference (2006)

23. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report 800-22, NIST (2001)
24. Sun, L., Versteeg, S., Boztaş, S., Yann, T.: Pattern Recognition Techniques for the Classification of Malware Packers. In: Steinfeld, R., Hawkes, P. (eds.) ACISP 2010. LNCS, vol. 6168, pp. 370–390. Springer, Heidelberg (2010)
25. Tabish, S.M., Shafiq, M.Z., Farooq, M.: Malware detection using statistical analysis of byte-level file content. In: Proc. ACM SIGKDD Workshop CyberSecurity and Intelligence Informatics (2009)
26. Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware. In: Proc. BlackHat DC Conf. (2007)
27. Wicherski, G.: peHash: A novel approach to fast malware clustering. In: USENIX Workshop Large-Scale Exploits and Emergent Threats, LEET (2009)

# Experiments with Malware Visualization*

Yongzheng Wu[1] and Roland H.C. Yap[2]

[1] Singapore University of Technology and Design
20 Dover Drive, Singapore 138682
yongzheng_wu@sutd.edu.sg
[2] School of Computing, National University of Singapore
13 Computing Drive, Singapore 117417
ryap@comp.nus.edu.sg

**Abstract.** This paper proposes DotPlot visualizations [1,8] for comparing and clustering malware. We describe how to process and customize the malware memory images to get robust and scalable visualizations. We demonstrate the effectiveness of the visualizations for analysing, comparing and clustering malware.

## 1 Introduction

Malware is often analysed using static or dynamic analysis techniques. Malware is also classified into families of related or close variants. However, the malware comparison/classification/clustering problem is not straightforward [2]. Antivirus scanners may classify the same malware instance as belonging to different families. This shows that the ground truth may not be so clear. In this paper, we take an orthogonal approach – using visualization to aid other techniques.

There are few works on visual analysis of malware. Nataraj et al. [4] visualize binaries by representing each byte using a gray scale pixel and apply image processing techniques such as texture extraction to get features from the image which are used for malware classification. Panas [5] visualizes each function in a malware binary as a point in 3D space according the function's statistics. The points form into a landscape metaphor, which can be visually compared. Other work [6,7] visualize the dynamic behaviour of malware.

In this paper, we experiment with DotPlots as a means for comparing the similarity between malware. DotPlots are used to compare similarity in sequences. It has been used for self-similarity comparisons in music [1] and for comparing DNA sequences [3]. We based our experiments on `lviz` [8][1] which allows extensive customization of DotPlots. Our goal is to be able to analyse, compare and cluster malware instances through the use of visualization. As malware analysis is not a problem with only a single answer, visualization is meant to be complementary to other automated techniques and can help to support or point out problems with other analysis.

* This work has been supported by grant R-394-000-054-232.

[1] `lviz` was originally designed for visualizing (Windows) operating system traces [9].

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 123–133, 2013.

Our experiments show that similarities in unpacked memory images of malware instances from the same family can be easily visualized. The hooking behavior of malware can be seen, which may motivate more detailed analysis. We show that an overall visualization of malware families (5 instances per 10 families: 142MB) can identify similarities and differences both intra and inter family. Visualization can also identify unknown malware.

## 2 Visualization

We begin by illustrating the malware visualization experiments with an example. Fig. 1 visualizes two variants of the Bagle worm, TrojanDownloader:Win32/Bagle.QM (worm A) and TrojanDownloader:Win32/Bagle.RL (worm B). The goal is to be able to see at a glance if worms A and B are similar/related and also where this occurs. The visualization used is called a *DotPlot* (DP) [1,8]. Black pixels show similarity between the contents of memory in each worm. The x and y axis represent (in a fashion) the contents of memory in worm A and B ordered by memory address. A black dot is drawn at position $(i, j)$ if the memory contents of worm A at address $i$ (on the x-axis) matches that of worm B at address $j$ (on the y-axis), otherwise, a white dot is drawn. In this paper, we annotate figures with ellipses, number labels and arrows – these are not in the original visualization and are markups for explanatory purposes. For the moment, we focus on the region with label 2 – it shows the DotPlot comparison between the executable memory sections in both worms. The dark diagonal region indicates strong similarity between the unpacked memory coming from the EXE (the executable file) which shows that worm A and B are related.

We use `lviz` [8] which provides interactive exploration of extended DP visuals. Each axis of the DP represents a linearly ordered sequence of tokens. The x and y-axis is shown by the red Labels 5 and 6. The rectangular region bordered by the x and y-axis is the DotPlot. To the right of the y-axis (Label 5) is a vertical strip (respectively, below the x-axis, a horizontal strip), called the *barcode*, which shows a customizable property about its respective sequence. `lviz` allows customization of the matching between the $i$-th token on the x-axis with the $j$-th token on the y-axis. The visualization is interactive with zoom in/out and different ways of normalizing the images. Consider two sequences of $n$ tokens, its DP visual is a virtual image with $n^2$ pixels, i.e. the DP of two 500K long sequences is a virtual 250 gigapixel image. `lviz` efficiently handles real-time interaction and display of sequences with lengths on the order of several 100K. Fig. 2 illustrates interactive exploration, starting with the initial DP in Fig. 2a, interactive zoom-in at the region indicated by the arrow shows progressively more detail going all the way to the finest detail DP showing single pixels and tokens in Fig. 2e. Notice that zoom-in is not the same as magnification as `lviz` equalizes the image so that it can better highlight small matching areas in the DP versus large areas.

In this paper, `lviz` is customized for malware visualization as follows. The sequences of tokens used on a DP axis are either consecutive bytes of memory

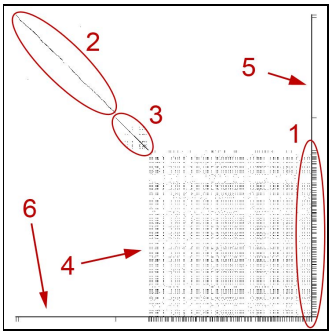



**Fig. 1.** Comparing two variants of Bagle

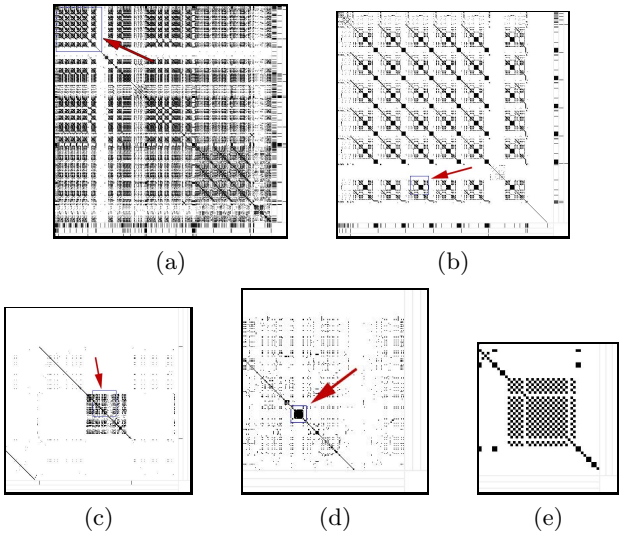



**Fig. 2.** Interactive Zooming In. (b) corresponds to the rectangle area in (a) highlighted by the arrow. (c) corresponds to the rectangle area in (b), etc.

or a sequence of $n$-grams at the corresponding virtual address. The sequence of $n$-grams need not come from consecutive virtual addresses but are ordered in ascending virtual address. The barcode is used to indicate: (i) a particular memory section (inner-most and closest to the axis); (ii) which sections come from a particular executable/DLL; and (iii) which executables are in a particular malware family (outer-most part). Matching is straightforward, namely, comparing the value of bytes or $n$-grams.

### 2.1 Section Extraction and Processing

We now describe how we obtain and process the token sequences.

**Dumping Sections from Memory**: As malware is usually packed, we use memory dumps instead of the file binary. Our visualization works on the memory

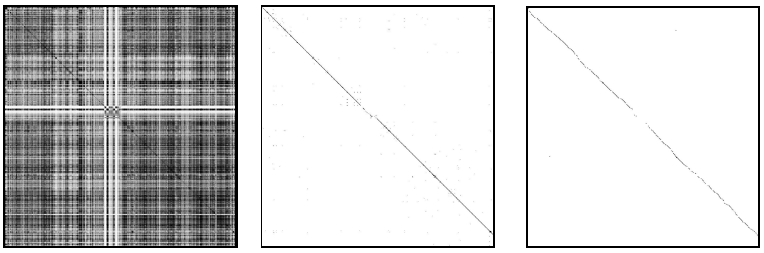
through unpacking the malware, any reliable technique for this can be used. The virtual memory of a process consists of many sections, each of which is a memory region with the same page protection (read, write and execute) and a file mapping. A section can be mapped from a file (mapped section) or created dynamically (anonymous section). Code in the EXE and dynamic link libraries (DLL) are in mapped sections. Sections such as stack and heap are anonymous sections. In this paper, we are mainly interested in code, so we only dump executable sections, but they may contain data as well. Many techniques can be used to extract the memory sections, in the experiments we use a simple heuristic for dumping memory.[2] However, the visualizations are not dependent on the extraction technique and any effective techniques can be employed.

**Selecting Sections**: We can select a subset of the dumped sections to visualize. Since we are interested in the malware, we want to exclude sections which come from system DLLs. To do this, we use the above method to extract executable sections from various "benign software" to obtain a collection of sections which we call *benign sections*. Sections in malware that have exactly same content as one of the benign sections are removed. In this paper, we call the remainder sections as *non-benign* (not belonging to the benign set). Further section removal criteria are described in the specific experiments.

***N*-Gram Generation**: A direct DP visualization of memory at the byte level gives too much matches to be useful, see Fig. 3a. Some reasons are: (i) runs of repeated values, e.g. consecutive no-op instructions for alignment padding; and (ii) higher level similarity, such as instructions or functions, occurs less frequently than byte level similarity. Rather than using the contents of memory, we first reduce consecutive bytes of the same value into one byte, then from a sequence of $m$ bytes we create a sequence of $m - n + 1$ $n$-grams. We found 16-gram to be reasonable for this purpose. Fig. 3b shows the 16-grams DP from the executable section of Bagle (Label 2 in Fig. 1)).

**Hash-Based (Content) Sampling**: We want to be able to deal with large malware sizes and concatenations of many malware into one sequence. However, the DotPlots would be far too large to be practical, thus, we want smaller sequences. Sampling the sequence at random positions (or any position-based method) to get a smaller one can fail as mis-alignment can cause sequences which match exactly to no longer match. We apply a content-based sampling method, i.e. the sampling decision is based on the content. We randomize content selection by choosing a range of values from the hash. From a sequence of $m$ $n$-grams, we reduce this to approximately $O(m/k)$ $n$-grams (assuming a uniform hash distribution) by selecting $\frac{1}{k}$ of the hash space, e.g., compute a 64-bit hash for each

[2] We run the malware for a pre-determined period of time so that its unpacking and online code downloading completes. If it has not quit yet, we then use `VirtualQueryEx()` to iterate through its virtual memory and `ReadProcessMemory()` to dump all executable sections. In our experiment, we run all malware for 5 seconds. Only 21% malware samples quit during this period and we do not consider those samples in our experiments.



(a) (b) (c)

**Fig. 3.** The effect of $n$-gram and hash sampling

**Table 1.** Simple polymorphic code in Bagle

| address | Bagle 1 | | Bagle 2 | |
|---|---|---|---|---|
| | opcode | instruction | opcode | instruction |
| 004013c1 | 68c0204500 | push 0x4520c0 | 68c0204500 | push 0x4520c0 |
| 004013c6 | 90 | *nop* | e8c6055402 | *call 0x2941991* |
| 004013c7 | e8f1045402 | *call 0x29418bd* | 90 | *nop* |
| 004013cc | ff15c0204500 | call [0x4520c0] | ff15c0204500 | call [0x4520c0] |
| 004013e9 | 7505 | jnz 0x4013f0 | 7505 | jnz 0x4013f0 |
| 004013eb | e8af9a0100 | call *0x41ae9f* | e821a60100 | call *0x41ba11* |
| 004013f0 | 50 | push eax | 50 | push eax |
| 004013f1 | e8337a0300 | call *0x438e29* | e8a5850300 | call *0x43999b* |
| 004013f6 | cc | int3 | cc | int3 |

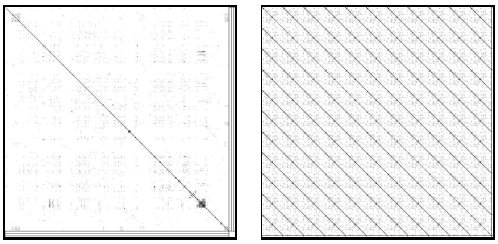
$n$-gram and keep the $n$-gram if its hash value is $< 2^{64}/k$. Fig. 3b is the hash sampled version of the full $n$-gram DP from Fig. 3b. It shows that while the length of the sequence is reduced by hash sampling, the overall visualization is still similar at $k = 500$.

## 3 Malware Visualization Applications

We now show various ways we can visualize malware. The experiments differ in the comparison objective and how sections are processed.

### 3.1 Visualizing Two Bagel Variants

We revisit the motivating example which is to compare two Bagle worm variants. Suppose we compare the DotPlot of both EXEs, the result is an almost white image with very little similarity. This is probably due to packing. Instead, Fig. 1 compares all non-benign sections from each worm. The barcode circled by Label 1 shows that there are many small sections. Their corresponding DotPlot (many dots forming a large square) shows that these small sections have similarities among the same variant, and also between the two variants. The diagonal line circled by Label 2 is broken into many tiny segments. This means that the code in both variants is mostly the same except many different instructions which are scattered across the whole section. Zooming in and clicking tells us more detailed information such as $n$-grams and section properties. We then investigated the two sections by disassembling them. Table 1 shows two code fragments where the two variants differ. These differences are probably caused by some form of



(a) (b)

**Fig. 4.** Similarity Among Modified DLL Sections. (a): Comparing the original `kernel32.dll` section with the modified copy by Hupigon. (b): Self-comparison of the original `ntdll.dll` section and 10 different modified copies by Conficker.

**Table 2.** API hooking in Hupigon. 0x7c801d7b is the entry of `LoadLibraryA()`. 0x7c8197b0 is the entry of `CreateProcessInternalW()`.

| | benign | | Hupigon | |
|---|---|---|---|---|
| address | opcode | instruction | opcode | instruction |
| 7c801d7a | 90 | nop | 90 | nop |
| 7c801d7b | 8bff | *mov edi,edi* | e9dd22c483 | *jmp 0x44405d* |
| 7c801d7d | 55 | *push ebp* | | |
| 7c801d7e | 8bec | *mov ebp,esp* | | |
| 7c801d80 | 837d0800 | cmp [ebp+0x8],0 | 837d0800 | cmp [ebp+0x8],0 |
| 7c8197af | 90 | nop | 90 | nop |
| 7c8197b0 | 68080a0000 | *push 0xa08* | e9079dc283 | *jmp 0x4434bc* |
| 7c8197b5 | 68889a817c | push 0x7c819a88 | 68889a817c | push 0x7c819a88 |

basic polymorphic code introduced to interfere with anti-virus software. There are about 5000 code fragments like this, which break the diagonal line circled by Label 2 into 5000 tiny segments. The straight diagonal line circled by label 3 shows that the corresponding sections are mostly the same.

### 3.2 API Hooking in Hupigon and Conficker

Malware may modify the behaviour of system library functions by API hooking. A common hooking method is to directly modify the function code. We can easily discover this by visualizing *unique* DLL sections. We remove sections which have identical content from all DLL sections to obtain only unique ones. Since hooking modifies very little code, the modified section will have strong similarity to the unmodified one.

Fig. 4a compares the original `kernel32.dll` section with the modified copy by the Hupigon trojan. We can conclude that the modified copy is very similar to the original because the diagonal line is straight and continuous. This leads us to compare the difference between the disassembly, i.e. using `diff`. We found only two modified code fragments, shown in Table 2 which hooks two system library functions. Fig. 4b is a self-comparison (x and y-axis represent the same sequence) with 10 sections modified from `ntdll.dll` in 10 variants of the Conficker worm concatenated with the original `ntdll.dll` giving a total of 11 unique sections Again we can conclude that they are all very similar, and here we find hooking of `NtQueryInformationProcess()`.

**Table 3.** Statistics of the malware collection used. All columns except for "total no. of samples" give the average per malware sample of the family.

| Family | total no. size of samples | no. of sections | no. of benign sections | no. of anon sections | non-benign size | no. of sampled $n$-gram ($k = 500$) |
|---|---|---|---|---|---|---|
| Alureon | 5 | 47.4 | 39.6 | 6.4 | 0.6MB | 0.6K |
| Bagle | 66 | 126.0 | 25.9 | 99.0 | 3.1MB | 5.3K |
| Bifrose | 46 | 36.9 | 16.7 | 18.1 | 2.4MB | 6.3K |
| Conficker | 42 | 29.5 | 23.4 | 5.5 | 0.8MB | 1.3K |
| Hupigon | 58 | 201.1 | 19.2 | 177.7 | 2.7MB | 2.6K |
| Ldpinch | 102 | 36.8 | 29.0 | 7.5 | 0.7MB | 0.3K |
| Midgare | 58 | 27.3 | 15.0 | 12.1 | 2.7MB | 5.3K |
| Mytob | 5 | 52.8 | 24.8 | 25.8 | 1.8MB | 2.9K |
| Mydoom | 50 | 27.0 | 21.8 | 4.0 | 0.3MB | 0.2K |
| Netsky | 8 | 35.2 | 29.8 | 4.4 | 2.5MB | 0.3K |
| Turkojan | 82 | 44.6 | 23.4 | 5.5 | 3.8MB | 9.8K |
| Zlob | 89 | 34.9 | 27.3 | 4.1 | 1.3MB | 3.7K |

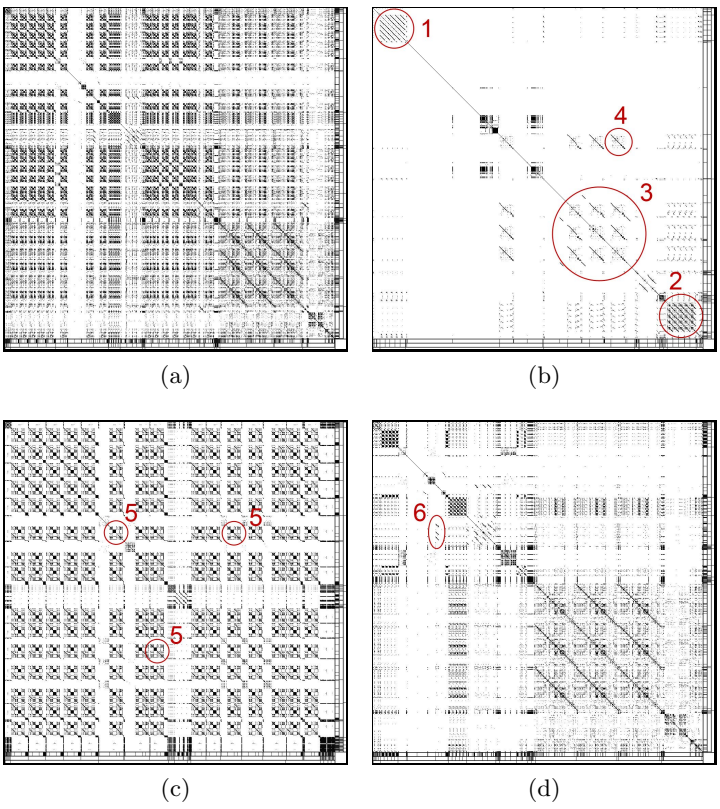
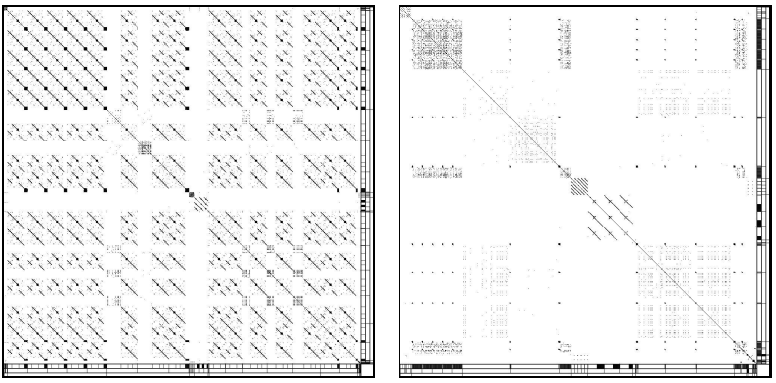



**Fig. 5.** Comparing 60 malware instances: 5 instances per 12 malware families (a): all non-benign sections; (b): only EXE sections; (c): only anonymous sections; and (d): only sections unique to the family.



(a) (b)

**Fig. 6.** The Effect of Filtering. We apply filtering on Fig. 5c. Fig. (a) shows filtering with low=10, high=100 and family=4. Fig. (b) is shows a more aggressive filtering with low=0, high=50 and family=2.

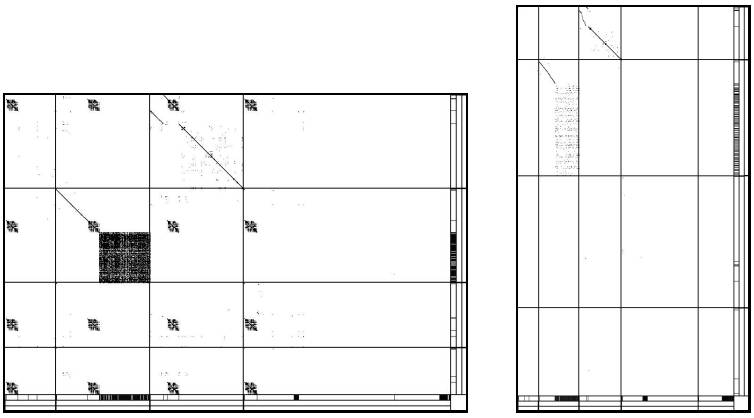
### 3.3 Visualizing Malware Families

We use a collection of 610 malware samples belonging to 12 families (see Table 3). The goal of this experiment is to see the similarities in malware belonging to the same family; and similarities across different families. We randomly selected 5 samples from each family for this experiment. Fig. 5 shows the self-comparison of the 60 malware samples concatenated together. There are four visualizations which differ in how sections are selected. The malware samples are ordered by their families following the order in Table 3, i.e. the Alureon family is at the left most (x-axis) and respectively top most (y-axis). Each axis has three barcodes. The inner barcode separates memory sections; the middle barcode separates malware samples; and the outer barcode separates families.

Fig. 5a contains all the non-benign sections with a total size of 142MB. Hash-based sampling reduces this to 194K $n$-grams. From the outer barcode, we can see that different malware families have different sizes, which conform to the "no. of sampled $n$-gram" column in Table 3. The sampling reduction ratio ("no. of sampled $n$-gram"/"non-benign size") is mostly between 1:400 to 1:1500[3], which shows that the reduction by hash-based sampling is sufficiently uniform. Fig. 5a also shows that there is considerable similarity across different families.

Fig. 5b consists only of EXE sections. Comparing the middle and inner barcode, we see that there can be several EXE sections per malware instance. Label 1, 3 and 2 show intra-family similarity of Bagle, Hupigon and Turkojan respectively. Label 4 shows an instance of Bifrose that is similar to some Hupigon instances. We then classified the Bifrose instance with a number of anti-virus software, and found that some of them consider it to be Hupigon, which is consistent with the visualization. Some anti-virus websites also list the two as

[3] Netsky is 1:8000, due to a large number of consecutive null bytes.



(a) (b)

**Fig. 7.** Identifying Unknown Malware. X-axis: 4 known malware samples (Alureon, Bagle, Conficker and Hupigon). Y-axis: 4 unknown malware samples. For easier viewing, we added separators for the samples. (a): not filtered; (b): filtered.

aliases. Fig. 5c contains only the anonymous sections. The pattern highlighted by label 5 shows that there are similar anonymous sections shared by some samples across 4 families. Fig. 5a shows even more inter family similarities which is less useful if we want to highlight the intra family similarities. Fig. 5d is obtained by removing all sections which appear in more than one family. The diagonals highlighted by Label 6 again show the similarity between Bifrose and Hupigon.

The similarity shared only between two malware samples is sometimes more important than the similarity across all samples. Selecting only unique sections as shown in Fig. 5d is one way of showing this type of similarity. Alternatively, we can look at the uniqueness at the $n$-gram rather than section granularity. We remove frequent $n$-grams or $n$-grams appearing in many families. In this experiment, we define 3 thresholds: $low$, $high$ and $family$. Suppose an $n$-gram appears $m$ times in $n$ families, we remove it if $m > high \vee (m > low \wedge n > family)$. Fig. 6 shows the results of filtering Fig. 5c with two different settings. We can see that some very frequent appearing patterns highlighted by Label 5 in Fig. 5c are mostly filtered, leaving only small diagonals in Fig. 6a. Some of those diagonals filter even more strongly in Fig. 6b, leaving mostly intra family similarities shown by the rectangular diagonal regions within family boundaries.

### 3.4 Identifying Unknown Malware

We can use similar techniques in Sec. 3.3 to identify unknown malware given a collection of identified malware. In Fig. 7, we put 4 known malware belonging to different families on the x-axis, and 4 unknown malware on the y-axis. This effectively gives a 4x4 cross comparison of each known to each unknown malware. The selected sections are unique to the 4 known families (similar to Fig. 5d). Fig. 7a shows the first unknown malware is most similar to Conficker and the

second unknown malware to be most similar to Bagle. The last two unknown malware samples show small and roughly equal similarity to all 4 families. An anti-virus scan on the 4 unknown malware classifies them as Confiker, Bagle, Turkojan and Bifrose, which is consistent with the visualization. If we apply a stronger intra-family filtering with low=0, high=50 and family=2, we have an even more obvious visualization in Fig. 7b.

## 4 Discussion and Conclusion

Address space layout randomization (ASLR) is commonly employed in many operating systems including Windows as a defence mechanism against attacks using the existing code. ASLR affects our visualization because randomization can change the contents of non-position independent code due to relocations, thus, introducing additional differences. This will affect our benign section filtering mechanism which uses exact matching. However, benign section filtering is mainly to remove sections from the visualization and a more sophisticated filtering can be used to do approximate matching. A simpler alternative is to turn off ASLR when benign section filtering is being used. Another alternative is to apply the inverse of the relocation on the extracted sections.

The next question is how ALSR affects the DotPlot visualization? ALSR only affects a small percentage of the instructions at most. This has very little affect on the overall DP. Looking at the "big picture", the similarity is still obvious.

We emphasize that in this work, we want to show the potential of visualization as an adjunct to existing analysis techniques which may be automated. Visualization, on the other hand, is meant for human consumption. Instead we want this to complement and help verify other malware analysis, e.g. malware analysis may decide that a malware belongs to family A but visualization can show that it is also similar to family B. The applications shown in Sec. 3.3 and 3.4 are thus not designed for analysing large amount of malware samples but rather to focus on selected malware comparisons.

In conclusion, our experiments demonstrate that visualization is effective in showing the similarity in the internal structure of malware through variants in the family. It also shows similarities between families. This may be due to the eco-system of malware creation and development. It may also explain why other manual/automatic classification techniques are successful.

## References

1. Foote, J.: Visualizing Music and Audio using Self-Similarity. In: ACM Multimedia (1999)
2. Li, P., Liu, L., Gao, D., Reiter, M.K.: On Challenges in Evaluating Malware Clustering. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 238–255. Springer, Heidelberg (2010)
3. Maizel, J.V., Lenk, R.P.: Enhanced Graphic Matrix Analysis of Nucleic Acid and Protein Sequences. National Acad. of Science (1981)

4. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware Images: Visualization and Automatic Classification. In: VizSec (2011)
5. Panas, T.: Signature Visualization of Software Binaries. In: SoftVis (2008)
6. Quist, D.A., Liebrock, L.M.: Visualizing Compiled Executables for Malware Analysis. In: VizSec (2009)
7. Trinius, P., Holz, T., Gobel, J., Freiling, F.C.: Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs. In: VizSec (2009)
8. Wu, Y., Yap, R.H.C., Halim, F.: Visualizing Windows System Traces. In: SoftVis (2010)
9. Ramnath, R., Sufatrio, Yap, R.H.C., Wu, Y.: WinResMon: A Tool for Discovering Software Dependencies, Configuration and Requirements in Windows. In: LISA (2006)

# Tracking Memory Writes for Malware Classification and Code Reuse Identification

André Ricardo Abed Grégio[1,2], Paulo Lício de Geus[2], Christopher Kruegel[3], and Giovanni Vigna[3]

[1] Center for Information Technology Renato Archer, Brazil
argregio@cti.gov.br
[2] University of Campinas, Brazil
paulo@las.ic.unicamp.br
[3] University of California, Santa Barbara, USA
{chris,vigna}@cs.ucsb.edu

**Abstract.** Malicious code (malware) is used to steal sensitive data, to attack corporate networks, and to deliver spam. To silently compromise systems and maintain their access, malware developers usually apply obfuscation techniques that result in a massive amount of malware variants and that can render static analysis approaches ineffective. To address the limitations of static approaches, researchers have proposed dynamic analysis systems. These systems usually rely on a sandboxing environment that captures the system calls performed by a program under analysis.

In this paper, we propose a novel approach to capture and model malware behavior that is based on the monitoring of the data values that a certain subset of instructions writes to memory during program execution. We have implemented a malware clustering component and a component to detect code reuse between different malware families. To validate our proposed techniques, we analyzed 16,248 malware samples. We found that our techniques produce clusters with high accuracy, as well as interesting cases of code reuse among malicious programs.

## 1 Introduction

Malicious software (malware) is a significant threat for cyber security. Current malware operations vary from stealing sensitive data to attacking critical infrastructures. Today's malware employs many different ways to propagate, including social engineering techniques to deceive a user to click on e-mail attachments, and drive-by download attacks that exploit web browsers and their plug-ins. In addition, obfuscation techniques are a powerful tool to render static malware analysis approaches ineffective and to decrease detection from signature based scanning. To address this problem, researchers have proposed dynamic analysis systems, which rely on the observed runtime activities (behavior) for detection and classification. To capture and model the behavior of malicious code, dynamic analysis systems typically rely on system calls. They treat the program as a black box and capture activity at a relatively high level. For example, two

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 134–143, 2013.

programs might be very different "inside" but might yield the same, visible effect to the "outside" by invoking the same system calls. While this might not be an immediate problem for malware detectors, it makes it hard to distinguish between different malware families.

To address the limitations of system-call-based detection and classification, this paper proposes a novel approach to capture and model program (malware) behavior. We record a trace that contains all the values that (a certain subset of) instructions write. These writes can go either to a destination register or a memory location. By looking at the intermediate data values that a computation produces, we analyze the execution of a program at a much finer level of granularity than by simply observing system calls. The main intuition is that by using the data values, we can produce a very detailed profile that captures the activity of individual functions. Also, data values are tied very closely to the purpose (semantics) of a computation, and, hence, are not as easy to disguise as the code that performs the computation. Malware authors have introduced many ways in which code can be altered so that syntactically different instructions implement the same algorithm (e.g., dead code insertion, register renaming, instruction substitution). However, when an algorithm computes something, we would expect that, at certain points, the results (and temporary values) for this computation hold specific values. Our goal is to leverage these values to identify (possibly different) code that "computes the same thing." The main contributions of this paper are the following: (I) we introduce a novel approach to capture and model behavior from dynamically analyzed malware that is based on the sequence of values that a program writes to memory or registers; (II) we describe a two-step procedure to decide whether two execution traces are similar and leveraged it to implement malware clustering and code reuse identification.

## 2 Data Value Traces

In this section, we discuss how we build *data value traces* to capture the activity of a (malware) program. To obtain these traces, we developed a prototype system that runs malware samples in an emulated environment. The prototype was developed using PyDBG [13]. This provides us with tight control of the debugging process. Also, PyDBG provides features to hide its debugging activity, which is useful to foil most malware attempts to detect the analysis environment.

We use our prototype to record an ordered sequence of instructions that modify at least one register or memory value; that is, we are only interested in instructions that *write* to memory. For each of these instructions, we store the numeric value(s) of all memory locations and registers to which this instruction writes (typically, this is one). For instance, if a malware sample executes the instruction `sub esp,0x58`, we will log in our trace the line `sub esp,0x58; 0x12ff58`, which corresponds to the instruction and the new value written to register `%esp`, which is 0x12ff58 in this example (assuming that the initial value of `%esp` was 0x12ffb0). When the malware process terminates or a timeout is reached, we also take a snapshot of the content of the malware's executable (code) segments.

This information is needed later to identify code reuse between malware samples, but it is *not* required to identify similarity between samples.

To collect the sequence of executed instructions, our system runs the sample in single-step debugging mode. More precisely, we single step through the code within the malware executable (and all code dynamically generated by the malware). However, calls that are made to standard operating system libraries are not logged, nor are the instructions executed inside these libraries. Fortunately, the system dynamically-loaded libraries (DLLs) are loaded into the memory region ranging from `0x70000000` to `0x78000000` (in Microsoft Windows XP). This speeds up the monitoring process and makes the resulting traces smaller. Also, it focuses the data collection on the actual malicious code.

To increase the efficiency of the collection process and to minimize the size of the traces, we only log a selected subset of instructions related to logic and arithmetic operations, namely: `add`, `adc`, `sub`, `sbb`, `mul`, `imul`, `div`, `idiv`, `neg`, `xadd`, `aaa`, `cmpxchg`, `aad`, `aam`, `aas`, `daa`, `das`, `not`, `xor`, `and`, `or`. We focus on these instructions because we are mostly interested in characterizing computations that the malware performs. Such computations will almost always involve arithmetic and logic instructions. Other instructions, such as data move or stack manipulation routines, are mostly used to prepare the environment for a computation, and hence, are less characteristic than the values that emerge directly as the result of a computation. We decided to remove the arithmetic instructions `inc` and `dec`, as they are typically involved in simple counters, which reveal little information about the data that is being computed. We also decided to remove instructions from the trace when they write the value 0, as this constant is not very characteristic of a particular computation.

We apply one last transformation to convert a sequence of instructions into the final data value trace. This transformation works by moving a sliding window of length two over the instruction sequence. For the two instructions in the window, we extract the two data values that these instructions write, one value for each instruction, and aggregate then into a pair of values – a bigram. After the bigram is appended to the data value trace, we advance the sliding window by one instruction. The reason for transforming the sequence of instructions (or written) values into bigrams is the following: If we would compare simple traces of individual values, it is more likely that two values in two traces match by accident. By combining subsequent values into pairs, we add a simple form of *context* to individual data values. We found that this extra context significantly lowers the fraction of coincidental matches and improves the separation between different program executions.

## 3 Comparing Traces

As discussed in the previous section, we capture the activity of a malware program by collecting a trace that consists of a sequence of bigrams of data values that this program has written. For a number of applications (such as malware classification and clustering), we require a technique to determine whether the

activities of two malware samples are similar. To perform this comparison, we have developed a two-step algorithm. This algorithm operates on two data value traces as input and outputs a similarity measure $S$ that ranges from 0 (completely different) to 1 (identical).

### 3.1 Step 1: Quick Comparison

The goal of the first step is to decide whether two traces are similar enough to warrant a further, more detailed comparison. This step works by creating a small "identifier" for each trace. This identifier is based on the $k$ least-frequent bigrams that appear in a trace. The underlying assumption behind this choice is that if two samples are variants of each other, they should share some specific features or attributes that are particular to their family. Thus, we can discard the most common bigrams, which can appear within many different families, and focus on the specifics of a certain family's fingerprint. We have experimentally determined that a value of $k = 100$ yields good results.

More formally, we state our approach as follows. Let $ID_{M_1}$ and $ID_{M_2}$ be the $k$ least-frequent bigrams from traces produced by malware samples $M_1$ and $M_2$. We compare these two malware identifiers by applying the Jaccard index ($J(ID_{M_1}, ID_{M_2}) = \frac{ID_{M_1} \cap ID_{M_2}}{ID_{M_1} \cup ID_{M_2}}, 0 \leq J \leq 1$). If, and only if, the Jaccard index (ranging from 0 to 1) is greater than the empirically established threshold of 0.3[1], we move to the second step. Otherwise, the result of this computation is used as the similarity value (which indicates low similarity).

### 3.2 Step 2: Full Similarity Computation

In the next step, we compute the overlap of the entire two traces. More specifically, we compute the longest common subsequence (LCS) between them. Suppose that $T_1$ and $T_2$ are different data value traces and that $L_1$ and $L_2$ are their lengths, respectively. The similarity between the two traces is then calculated as $C(M_1, M_2) = \frac{LCS(T_1,T_2)}{min(L_1,L_2)}$. We chose the longest common subsequence over the longest common substring to tolerate small differences in the computations. Moreover, we note that using a standard LCS algorithm can be computationally expensive. We addressed this by calculating the LCS based on the GNU `diff` tool (`http://en.wikipedia.org/wiki/Diff`) output. Our experiments, evaluating a standard LCS implemented in C++ and our approximate LCS computation showed that we could accomplish faster results using our approach — in some cases ≈500× faster — with no significant loss of accuracy.

The original `diff` tool has the nice property that it inserts "barriers" while computing the longest common subsequences present in a textual input. Our `diff`-based LCS approach, referred from now on as *eDiff*, enhances this capability by (i) marking the regions that differ between two traces and (ii) by mapping the shared subsequences to the original instructions in the respective execution

[1] To choose this threshold ($T$), we performed tests with an increment of 0.1 for the range $0.0 < T \leq 0.5$.

traces. As a result, we know exactly what malware code produced similar memory writes. This will be useful for identifying code reuse, as explained in Section 5.2. To map value traces back to instructions, we simply link the bigram values in the value traces to the raw instructions that produced those values.

## 4 Applications

In this section, we discuss two applications that we built on top of our malware trace similarity technique. The first application is clustering; the idea is to group samples that show similar activity based on their value traces. The second application uses the data value traces to find cases of code reuse. That is, we want to find cases in which malware samples that belong to different families share one or more snippets of identical code.

### 4.1 Clustering

The input to the malware clustering application are a set of $N$ data value traces, one trace for each of the $N$ samples to be clustered. The goal is to find groups of malware samples that are similar. Clustering is implemented in two steps: pre-clustering and inter-cluster merging.

**Pre-clustering:** The goal of the pre-clustering step is to quickly generate an initial clustering and avoid having to perform $N^2/2$ comparisons. To accomplish this, we sequentially process each of the $N$ samples, one after another (in random order), as follows: Each new sample is compared to all cluster leaders (explained below), using the similarity computation described in the previous section. When the trace for the new sample exhibits more than 70% similarity with one or more cluster leaders, this sample is merged with the existing cluster for which the similarity is highest. Otherwise, the sample (and its trace) is put into a new cluster, and this sample also becomes the cluster leader. When merging a trace with an existing cluster, we need to elect a new cluster leader (a cluster leader is basically the trace that is selected to represent the entire cluster). For this, we must make a selection between the existing cluster leader and the new trace. We select the *longer* trace as the new leader. We do this to increase the probability that a sample, whose behavior is similar to the activity of malware in a cluster, is properly matches with that cluster. In other words, by selecting the longest trace as the cluster leader, a new trace has more chances to find a long, common subsequence. By removing from the comparison computation all except one trace for each cluster, we greatly reduce the required number of comparisons.

**Inter-cluster Merging:** The pre-clustering step results in a set of initial clusters whose traces share at least 70% similarity. However, due to the nature of the quick comparison (first step of the similarly comparison), there can be clusters that should be merged but are not. That is, it is possible that two traces are actually quite similar, yet their least-frequent bigrams are too different to pass the threshold. In this case, there are different clusters containing malware from the same family, and it is desirable to merge these clusters. The merging step

is applied to the output of the pre-clustering step so as to generate a reduced amount of clusters. To this end, we perform a pairwise comparison between all cluster leaders, using our *eDiff* algorithm. If their similarity is greater than the same 70% threshold defined previously, the clusters are merged.

### 4.2 Code Reuse Identification

When comparing two traces, our algorithm not only computes a general similarity (overlap) score but also determines which parts of the traces are identical. When we find a stretch of values that are identical between two traces generated by executing different samples, we might naturally ask the question whether these values were produced by similar code. This would allow us to identify code that is shared between samples that are otherwise different.

To identify code reuse, when *eDiff* compares two data value traces, it stores for each element (bigram) in the traces whether this element is unique to the trace or shared between both traces. For instance, let us assume that we have two traces. One contains the three bigrams: `(0x1,0x2)`, `(0x2,0x4)`, and `(0x4,0x5)`; the other contains the four bigrams: `(0x1,0x2)`, `(0x2,0x7)`, `(0x7,0x4)`, and `(0x4,0x5)`. In this case, *eDiff* would find that the first and last element in each trace are shared, while the middle one(s) are unique (to each trace). To find code reuse, we check both traces for the presence of at least four consecutive elements that are shared. The threshold of four was empirically determined and allows us to find shared code roughly at the function level. A higher threshold would be possible when we want to find longer parts of shared code. A lower threshold often yields accidental matches that do not reflect true code reuse.

Next, we require a mechanism to "map back" values in a data value trace to the instructions that produced them. This can be done easily because we retain the original instruction sequences that were recorded during dynamic analysis. To find the code in the malware program that contains the "shared instructions" we generate a regular expression pattern, which is then matched against the dumped code segment. When a match is found, we consider the resulting code block as a candidate for reuse. All matches that are found for each trace are compared, and when we find a sequence of identical code of a minimum length, we identify the code snippet as reused between malware samples.

## 5 Preliminary Experiments

We performed an initial set of experiments using 16,248 execution traces that produced promising results. These traces were obtained from the analysis of Windows PE32 executable programs and they represent a diverse and recent set of different malware families that are currently active in the wild.[2]

### 5.1 Malware Clustering

The evaluation of the quality of a clustering algorithm is a complicated task [5], as clustering results are often not objectively right or wrong but depend on a

[2] For a complete list of MD5 sums of the samples, please contact the authors.

number of factors, such as the metrics used to calculate the distances among samples and clusters, the final amount of clusters generated, the chosen heuristics, etc. We used three different ways to obtain a reference clustering, based on [9]: one from the static analysis of malware [4], one from the dynamic analysis of malware [2], and the last one based on anti-virus (AV) labels from AVG (`http://www.avg.com`), Avira (`http://www.avira.com`) and F-Prot (`http://www.f-prot.com`). These AV labels were relabeled so that only the general identifier for each family remains (e.g., Trojan.Zbot-4955 became "zbot").

After we generated the reference clustering sets, we borrowed the precision and recall metrics from [2] to measure the quality of our clustering results. The product of the values obtained from the overall precision and recall can be used to measure the overall clustering **quality** ($Q = P \times R$). For the reference clustering based on AV labels, we measured the quality of our clustering scheme by defining the level of agreement related to the labels assigned to each sample in a cluster. Perdisci et al. [11] proposed to use two indexes (cohesion and separation) to validate their HTTP-based malware behavioral clustering. However, their approach "attenuates the effect of AV label inconsistency" due to the way the Cohesion Index is defined (there is a "gap" and a "distance" value that causes a boost in cohesion). To avoid this boost, we define a simpler level of agreement $A$ for a cluster $j$, calculated as:

$$A_j = \frac{\sum_{N \in AV} max_{lbl \in Labels_N}(frequency(lbl))}{[|T_j| * |AV|]}$$

where $AV$ is the number of AV vendors, $Labels_N$ is the set of the assigned labels and their related frequencies for each AV engine for each cluster ($N = avg, avira, fprot$), $T_j$ is the total amount of samples in the cluster.

**Reduced Dataset.** Before we applied our clustering technique to the entire dataset, we ran preliminary tests using a smaller subset, which consisted of 1,000 random samples. Those initial tests were important to experiment with and determine different threshold parameters. In particular, we varied the similarity threshold for the second step of the algorithm from 0 to 100% (incrementing by 10% after each iteration) and observed the highest quality, i.e., the average between the obtained static and behavioral quality values, for the similarity threshold of 70%. Moreover, the AV labels' level-of-agreement value for this threshold is also very high (0.894).

**Full Dataset.** Based on the results of the preliminary tests, we defined a similarity threshold of 70% for the *eDiff* process. We continued to use the initially-established Jaccard index threshold of 0.3 for the quick comparison. The amount of clusters produced by the two reference clustering sets for the 16,248 samples with traces were 7,900 clusters for the static approach and 3,410 for the behavioral one. Our approach produced 7,793 clusters that were compared to the reference clustering sets, generating the precision values of 0.758 and 0.846 and the recall values of 0.81 and 0.572 for the static and behavioral reference, respectively. Calculating the AV labels' level-of-agreement for our clustering yielded 0.871. These values yield average results of 0.843, 0.652, and 0.656 for precision, recall, and quality, respectively.

### 5.2 Code Reuse

To look for code reuse in samples that likely belong to different malware families, we only check pairs of malware clusters that are sufficiently different. More precisely, based on the clustering results obtained in the previous step, we look for pairs of clusters that have a similarity score between 10% and 30%. We identified 974 pairs of clusters that fulfill this requirement. We discovered 15 pairs (involving ten different clusters) that share code between them. More precisely, we found seven different blocks of code that seem to be reused among samples. We sent the ten representatives (one for each cluster) to VirusTotal (`http://www.virustotal.com`). Looking at the results, we noticed that the most common label assigned to them refers to different Trojan malware that all seem to attack online games (and, apparently, shared code to do so).

## 6 Related Work

Dynamic malware analyzers, such as [8] and [15], operate at the system call level and currently do not log the low-level values of an execution (memory and registers). Ether [3] performs both instruction and system call tracing to analyze malware in a transparent way by using hardware virtualization extensions, but it has several of prerequisites on the type of operating system, architecture and platform, which can limit its use. Indeed, we can divide malware classification techniques according to how the traces were obtained — i.e., through either static or dynamic analysis — and to the type of behavior gathered — i.e., either lower-level or assembly-related data or higher-level or system call information.

**Static Analysis Approaches.** Shankarapani et al. [14] propose two detection methods to recognize known malware variants without the need of new AV signatures: SAVE, which generates signatures based on a malware sample API calls sequence through the static analysis of its executable, and MEDiC, in which the signature of a malware sample is part of its disassembled code. SAVE performs an optimal alignment algorithm before applying similarity measure functions (cosine, extended Jaccard, and Pearson correlation). This kind of algorithm does not scale well if there are too many sequences or if the sequences are very large. For files whose size is among ≈500 Bytes to ≈1000 Bytes, the detection time can be in a range of few seconds (considering just one executable). Kinable and Kostakis [7] performed a study of malware classification based on call graph clustering, where they measured the similarity of call graphs that were extracted from malicious binaries through matches that try to minimize the edit distance between a pair of graphs. The authors conclude that it is difficult to partition malware samples in well-defined clusters using $k$-means based algorithms, chosing the DBSCAN algorithm to cluster some sets, the larger having 1,050 samples. They state that this larger set had 72% correct clusters. Zhang and Reeves [16] propose a method to detect malware variants that uses automated static analysis to extract the executable file semantics. These semantic templates are characterized based on the system calls executed by a malware

sample and used in a weighted pattern matching algorithm that computes the degree of similarity between two code fragments.

**Dynamic Analysis Approaches.** Park et al. [10] present a classification method that uses a directed behavioral graph extracted from system calls through dynamic analysis. The generated directed graphs from two malware samples are compared computing the maximal common subgraph between them and the size of this subgraph is used as the similarity metric. Bailey et al. [1] developed a method based on the behavior extracted from a malware sample after executing it in a virtualized environment. This behavior is considered the malware's fingerprint and represents the set of actions that changed the system state, such as files written, processes created, registry keys modified, and network connection attempts. The tests were performed on a dataset of 3,700 samples from which the fingerprints were extracted. The normalized compression distance (NCD) was used as a similarity metric, and the pairwise single-linkage hierarchical clustering algorithm was used for classification purposes. Rieck et al. [12] propose the use of machine learning techniques on malware behaviors composed of changes that occurred in a target system in term of API function calls. They ran their experiments on more than 10,000 malware samples divided into 14 families labeled by AV software. The behavioral profiles obtained from dynamic analysis serve as a basis to feature extraction and use the vector space model and the bag of words techniques. After that, the Support Vector Machines method is applied to the feature sets for classification purposes. Bayer et al. [2] present a scalable clustering approach to classify malware samples based on the behavior they present while attacking a system. Dynamic analysis is used to generate the behavioral profiles — sequences of enriched and generalized information abstracted from system call data. The similarity metric used in their work is the Jaccard index, which is then used as an input to the LSH clustering method. Very recently, Jang et al. [6] introduced BitShred, an algorithm for fast malware clustering. In this paper, the authors present a new way to efficiently simplify and cluster features from inputs such as (static) code bytes and (dynamic) system call traces.

## 7 Conclusions

In this paper, we empirically demonstrated that the values stored in memory and registers after write operations can be used to detect and cluster malware in families. We also presented a different approach to perform the similarity score calculation that is simple and effective when applied to the malware problem. We compared the results from more than 16 thousand malware samples executed and processed in our prototype system to three reference clustering sets — static, behavioral (dynamic), and AV labeling — and our produced clustering reached an average precision value of 0.843 for the first two sets and a level of agreement value of 0.871 for the last one. Finally, we showed that our classification process can also be used to verify for code reuse, which helps to investigate the sharing of functions in different families of malware.

## References


1. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated Classification and Analysis of Internet Malware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 178–197. Springer, Heidelberg (2007)
2. Bayer, U., Milani Comparetti, P., Hlauscheck, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: 16th Symposium on Network and Distributed System Security, NDSS (2009)
3. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 51–62 (2008)
4. Jacob, G., Neugschwandtner, M., Comparetti, P.M., Kruegel, C., Vigna, G.: A static, packer-agnostic filter to detect similar malware samples. Tech. Rep. 2010-26, UCSB (November 2010)
5. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. 31, 264–323 (1999)
6. Jang, J., Brumley, D., Venkataraman, S.: BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In: ACM Conference on Computer and Communications Security, CCS (2011)
7. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. J. Comput. Virol. 7(4), 233–245 (2011)
8. Kruegel, C., Kirda, E., Bayer, U.: Ttanalyze: A tool for analyzing malware. In: Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference (April 2006)
9. Neugschwandtner, M., Comparetti, P.M., Jacob, G., Kruegel, C.: Forecast: skimming off the malware cream. In: Proc. of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 11–20. ACM (2011)
10. Park, Y., Reeves, D., Mulukutla, V., Sundaravel, B.: Fast malware classification by automated behavioral graph matching. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, CSIIRW 2010, pp. 45:1–45:4. ACM, New York (2010)
11. Perdisci, R., Lee, W., Feamster, N.: Behavioral clustering of http-based malware and signature generation using malicious network traces. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI 2010, p. 26 (2010)
12. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and Classification of Malware Behavior. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 108–125. Springer, Heidelberg (2008)
13. Seitz, J.: Gray Hat Python: Python Programming for Hackers and Reverse Engineers. No Starch Press, San Francisco (2009)
14. Shankarapani, M., Ramamoorthy, S., Movva, R., Mukkamala, S.: Malware detection using assembly and api call sequences. J. Comput. Virol. 7, 107–119 (2011)
15. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy Magazine 5(2), 32–39 (2007)
16. Zhang, Q., Reeves, D.: Metaaware: Identifying metamorphic malware. In: Proc. of the 23rd Annual Computer Security Applications Conference, ACSAC 2007, pp. 411–420 (December 2007)

# System-Level Support for Intrusion Recovery

Andrei Bacs, Remco Vermeulen, Asia Slowinska, and Herbert Bos

Network Institute, VU University Amsterdam
{abs204,rvn270,asia,herbertb}@few.vu.nl

**Abstract.** Recovering from attacks is hard and gets harder as the time between the initial infection and its detection increases. Which files did the attackers modify? Did any of user data depend on malicious inputs? Can I still trust my own documents or binaries? When malcode has been active for some time and its actions are mixed with those of benign applications, these questions are impossible to answer on current systems. In this paper, we describe DiskDuster, an attack analysis and recovery system capable of recovering from complicated attacks in a semi-automated manner. DiskDuster traces malcode at byte-level granularity both in memory and on disk in a modified version of QEMU. Using taint analysis, DiskDuster also tracks all bytes written by the malcode, to provide a detailed view on what (bytes in) files derive from malicious data. Next, it uses this information to remove malicious actions at recovery time.

**Keywords:** Attack recovery, dynamic taint analysis.

## 1 Introduction

We describe DiskDuster, a semi-automated system to help recover from intrusions. Intrusions may result from remote attacks, open network shares, exploits (Conficker [22]), user-installed Trojans (some versions of Torpig [27]), etc. However it spreads, the malicious code may interfere in deep and involved ways with the system state and removing the infection and its effects is difficult. For instance, Torpig turns off anti-virus scanners, modifies data, steals confidential information, and downloads/installs more malware on the victim's computer. Other attacks destroy data, or encrypt files for ransom.

Our recovery procedure aims to return the system to a sane state, as existed just before the attack, while retaining as much of the recent user data as possible. We show that we can undo most of the effects of complicated attacks. As an example, we demonstrate the usefulness of our approach for drive-by-downloads that fetch and execute malware that subsequently modifies the registry, and infects other programs that, in turn, modify system state. And so on. See Figure 2 for a full description of our running example. We evaluate our solution with several real attacks on Windows.

*Recovering from attacks.* Despite a plethora of defense mechanisms, attackers still manage to compromise computer systems. Sometimes they do so by corrupting memory and injecting a small amount of shellcode to download and install the real malware. Sometimes the users themselves install trojanized software. To make matters worse, the malware may be active for days before it is discovered.

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 144–163, 2013.

Upon discovery of a compromised machine, one of the most challenging questions is: what did the malware do? Which files has it modified? Did the attackers change or corrupt my financial records? Can I still trust any of the files created after the compromise, or should I check each and every one manually? Did the initial attack spread to other programs? And, most importantly, can I *undo* the malicious actions and restore the system to a sane state, without losing my recent data?

Currently, the only sane state a system can revert to is the last known good backup. This leaves the question of what to do with the changes to the system that occurred since then. Ignoring them completely is safe, but often unacceptable—losing valuable data generally is. Accepting them blindly is easy, but not safe—modifications may be the result of the malware's actions. However, the alternative of sifting through each of the files (or even blocks) on disk one by one to see whether it can still be trusted may be too time-consuming. Thus, we developed DiskDuster to automate most of this process.

*High-level overview.* Figure 1 illustrates DiskDuster's main flow of operations. The circled numbers in the text below correspond to the numbers in the figure. To minimize the performance impact, and to retain as much of information about the attack as possible, we decouple the analysis and recovery from the production machine. Thus, DiskDuster records the execution on the live production machine ① and replays it ② on a dedicated security server with additional security checks and recovery operations ③.

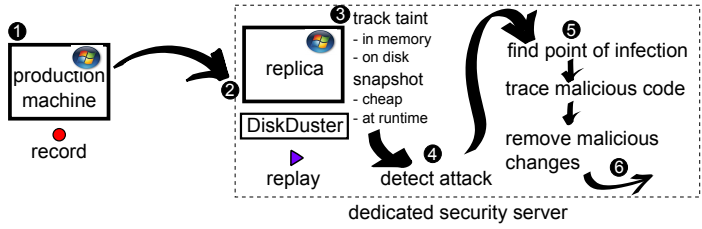



**Fig. 1.** Intrusion recovery in a decoupled security model

To recover the user data after an infection, we assume the presence of at least one detection method ④. The nature of the detection method is not important. The prototype in this paper works with dynamic taint analysis (DTA) and AV scanning, but we can easily add system call analysis or other techniques.

As soon as DiskDuster detects an intrusion in the replay, the user shuts down the original machine, while the security server continues to replay the trace, using DTA to monitor all the malcode's actions ⑤, tainting all writes by the malcode to memory and disk as malicious. Taint propagates whenever the malicious bytes are read, copied, or used in ALU operations. If malicious bytes compromise other processes, DiskDuster traces those also. Finally, DiskDuster cleans up the system by replaying benign disk writes up to the moment of infection. For the time between the infection moment and the detection moment, DiskDuster classifies all disk writes as 'benign' (not affected by the attack), 'malicious' (written by a malicious process) and 'suspicious' (possibly affected by the attack). Only suspicious data requires manual intervention.

*Contributions.* Most existing intrusion recovery approaches assume that the infection *cannot spread to the kernel* itself [15,3,14]—a very strong assumption that typically does not hold in practice. Others provide very limited protection (e.g., modification, but not removal, of system state and a few system files only [20]), or require users to define trusted data and malware manually [3]. Finally, existing approaches are typically tied to specific operating systems (often Linux, to have access to source code) [11,28,14].

In contrast, DiskDuster operates at the level of the (virtual) hardware and the approach can be applied to any OS. Throughout this paper, we focus on Windows, as it is (still) easily the most popular attack target. In addition, DiskDuster protects both the kernel and user processes and handles modification and removal of any file.

Thus, the contribution of this paper is an intrusion analysis and recovery system on top of a hardware emulator that works with *unmodified* OSs and applications and protects both kernel and user processes against complicated attacks. Our goal is to recover user *data*, but the system helps to recover other files and folders also.

Moreover, where modern tainting systems typically detect or track an attack on a single process, DiskDuster tracks the attack and all related processes, as well as their spread throughout the system. For instance, we track all disk writes of the malicious code, and take appropriate action when a benign process reads such bytes. Likewise, we treat processes that are started by a malicious process as malicious also. The same is true for threads injected by malicious code in a benign program. We are not aware of other systems with the same comprehensive tracking of malicious activity.

Tracking infections requires tracking the actions and data generated by the attack. Specifically, we need to know where this data ends up and what actions and data depend on it. Where almost all state-of-the-art intrusion recovery solutions [14,20] construct dependency graphs explicitly, DiskDuster tracks dependencies directly, by means of dynamic information flow tracking (taint analysis) and at byte-level granularity. Doing so is simpler and potentially weaker. But as it requires very little knowledge of the OS, it enables us to (a) support different OSs, and (b) handle kernel infections also. Moreover, we will see that the way DiskDuster handles implicit flows is very simple and yet very powerful. It allows it to limit taint tracking to explicit flows during analysis, while not losing even a byte of implicitly modified data (although overtainting may well occur).

Clearly, recovery cannot be complete if the attack had side effects beyond this system. For instance, if the malware sent spam, or leaked information to an external party, there is no way to undo this. We do revert changes on the file systems. We think this is sufficient for cleaning up infections locally. Even if some (memory-resident) attacks do not themselves leave any presence on disk, this is not a problem for DiskDuster. As long as it can detect the attack (e.g., using taint analysis), it will remove all disk writes that the malware influenced, while the malware itself will disappear after the reboot.

## 2 Threat Model and Assumptions

The ideal intrusion recovery system, upon detecting an attack, removes all harmful actions related to the attack automatically, leaving only changes to the system unaffected by the attack. Fundamentally, this is not possible—at least not in the general case. For instance, after an attack deletes the AV binary, a legitimate user may write a memo:

**A nasty attack**

The effects of real-world attacks like Torpig and Conficker have been complex and devastating. In this paper, we combine in our running example the effects of these and a number of others to create a complicated attack:

1. A drive-by-download infects the browser.
2. The attack immediately migrates to another running process on the same machine—infecting this process also. The migration complicates the tracking, since the second process did not connect to a malicious website.
3. The attack deletes the antivirus program.
4. Next, the shellcode in the second process downloads and executes the real malware and adds a registry key to make itself persistent across reboots.
5. Later, the malware encrypts the 'Documents' folder on disk, for ransom purposes, while deleting itself to prevent security experts from reverse engineering it.

**Goal:** to clean up the system and remove all traces of the attack.

**Fig. 2.** Attack scenario used as a running example

"No AV scanner present". Automated recovery may restore the AV scanner, but cannot spot the relation with the memo, resulting in inconsistencies (see also Section 5).

In practice, however, (semi-)automated recovery can be a powerful tool in post-hoc sanitization. By tracing what data was directly or indirectly generated by the attack, we reduce the load on the administrator significantly. We do not claim that DiskDuster is perfect. While it represents a significant improvement over the state of the art, and often restores systems automatically, we require human intervention in some cases. Still, even here DiskDuster indicates in detail which (parts of) files need further scrutiny.

*Assumptions.* In this paper, we assume the following:

1. Intrusions occur at arbitrary points in time and may not be detected until later.
2. Attacks can infect both user processes and the kernel.
3. Attacks may hide themselves root-kit style and turn off AV scanners and other defensive mechanisms on the guest OS.
4. DiskDuster can detect the attack and trace it back to the moment of infection. Given a recorded execution trace, we believe this is a reasonable assumption. A rootkit may hide itself, but it cannot remove itself from the execution trace, which means that AV scanners, taint trackers and other detection methods have a chance to detect it eventually. Once an AV scanner detects a trojan on the system, we skip backwards through the trace until we find a snapshot without the trojan binary, and then replay the execution until it is created and executed for the first time.
5. Attacks cannot tamper with the recording process undetected. As the recorder runs at the level of virtual hardware, this is a reasonable assumption.

*Decoupled security.* While it is *possible* to run DiskDuster on a stand-alone system, we designed it for decoupled security [4]. Decoupled security records the execution on a live production machine and replays it on a dedicated security server with additional

security checks and recovery operations (see Fig.1). In other words, all security checks and recovery operations run on the server.

Decoupled security hides the overhead of security checks from the production system. At a small, constant cost of recording on the production system, we can apply *any* security check on the replay side, including those too expensive to run on production systems. Since we use full-system DTA, the overhead of our analysis is very high (about 20x), we prefer to run it devolved from the production machine. Also, it is not possible for malware to hide 'rootkit-style' after the infection. As the initial point of infection is preserved in the execution trace, it can be found by periodic rescanning of older traces with new signatures. Finally, recording provides automatic backup, fine-grained versioning, and audit trails. Not surprisingly, decoupled security has become a popular security model [8,4,24,33]. Moreover, vendors like VMWare now offer record and replay functionality in their products [31].

Of course, recording and storing execution traces is not free, but in practice, the costs are low (a few percent increase of CPU overhead and minimal log sizes [4,24]). A more serious drawback of decoupled security is that attacks are always detected *a posteriori*. The same is true for traditional AV scanners. If a new trojan comes out, it takes a while before AV databases contain a signature for it. In either case, the challenge is to clean up the system and remove all traces of the attack.

*Implicit flows.* One of the most difficult problems for dynamic taint analysis is that of implicit flows [2,26], and we do not pretend to solve it in this paper. An implicit flow occurs when an assignment depends on a tainted value in a condition. For instance, consider the following code:

```
int y=0; if (x==1) y=1;
```

If `x` is tainted, perhaps `y` should be tainted also? After all, its value is completely determined by `x`. The problem is that implicit flows often lead to overtainting [2,26]. Recent work by Kang et al. [13] presents an interesting approach to curtail overtainting for certain applications, but for now implicit flows cannot be handled reliably. We do not try to solve them at all, but we cannot afford to ignore them either, as skipping them leads to false negatives. DiskDuster simply takes a conservative approach for malicious data on disk; whenever a process has read malicious or suspicious bytes, all subsequent writes are marked 'suspicious'. As a result, taint laundering is impossible. We discuss more interesting/problematic scenarios related to implicit flows in Section 5.

## 3 Architecture

After detecting an attack, DiskDuster traces it back to find the point of infection. It then uses DTA to track the malicious code's actions and undo the malicious effects. DiskDuster tries to remove these effects by restoring the disk to a pre-infection state, while presenting a user with lists of files and folders that became malicious or suspicious in the period between the attack and the detection. While the user can safely assume that files classified as benign are intact, they need to scrutinize the suspicious ones. Refer to Fig. 3 for a timeline illustrating the course of actions performed by DiskDuster.

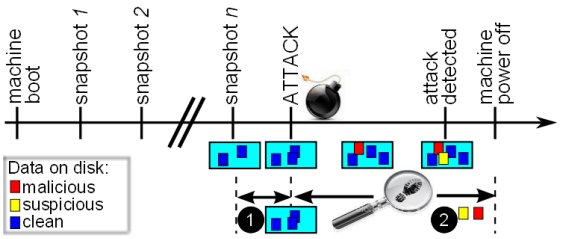



**Fig. 3.** DiskDuster timeline. Upon detecting an attack, DiskDuster restores the disk to a pre-infection state ①, removes all malicious data, and presents a list of all suspicious files/folders ②.

In addition, DiskDuster supports investigators by analyzing the attack. For instance, for drive-by-downloads, DiskDuster separates the shellcode from its packers, and when the shellcode downloads malware, it traces what bytes on disk change.

### 3.1 Decoupling DiskDuster: Recording and Replaying Execution Traces

Recording and replaying executions is hardly novel. In our lab, we have implemented and written about several such systems ourselves, both at full-system [10] and process [24] granularity. Others built similar solutions [8,1,30,19,33]. Moreover, VMware Workstation 6.5 introduced replaying as standard feature.

By recording only a minimum of non-deterministic events, the overhead of recording is small both in speed (a few percent) and storage (a few hundred Bps) [4]. Moreover, even with expensive detection methods like DTA, the *lag* between the original execution and the replica is minimal. In fact, the replayer typically has no problem keeping up with the recorder, mainly because it does not need to wait (e.g., for reads from the network or file system, or in idle loops). This is known as 'idle boost'.

While the best fit for DiskDuster is clearly our tailor-made Qemu-based full system replayer [10], we believe that with some effort other recorders, including VMWare's could be used also. Indeed, VMWare showed that one can record on VMWare and replay on Qemu in Aftersight [4]. In this paper, however, we focus on recovery.

### 3.2 Tracking, Logging, and Snapshotting

Figure 4 illustrates the DiskDuster components. We briefly enumerate each of the modules here, and describe them in more detail in subsequent sections. All these modules operate at the level of the emulated hardware and work with unmodified OSs.

*Tainting.* At the core of our architecture is a dynamic **taint tracking** module, capable of tracking data in memory and on disk. The module is based on Argos [23] and the propagation rules are similar to those of TaintCheck [18] and Minos [6]: (a) taint propagates to the destination (register or memory) whenever tainted data is copied, or used as a source operand in an arithmetic operation, (b) we clean the destination whenever an operation has a constant output (i.e., the output does not depend on the instruction's inputs), and (c) like most systems, we do not propagate taint on dereferences of tainted pointers.

Taint tracking in DiskDuster serves two different purposes. First, we use it just like most other DTA solutions—to detect control flow diversion and code injection attacks. For instance, DiskDuster taints all data coming from the network and raises an alarm whenever such bytes modify the control flow of the program directly (e.g., by overwriting the return address). Since we do not track indirect flows, DiskDuster may miss attacks that corrupt memory by means of bytes propagated through indirect transfers, be it tainted dereferences in translation tables, or implicit flows in conditions. It simply means that DiskDuster is not a pefect detector, but we do not see this as a serious limitation. We can easily complement DiskDuster with other detectors, such as AV scanners, but perhaps also more powerful taint trackers. After we detect an attack, the implicit flow is no longer a problem, because we conservatively track everything that could be influenced by malicious data.

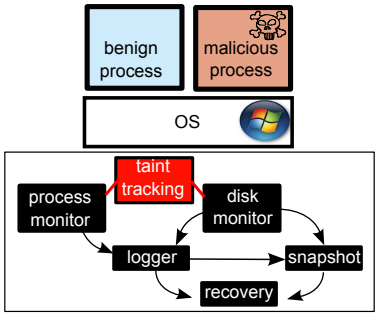



**Fig. 4.** DiskDuster architecture

Second, and much more essential, is that DTA allows us to monitor malicious programs. For instance, once we know that a process is malicious, we mark all bytes written by the process as malicious also—until we reach the end of the execution trace. Doing so allows us to separate good data from bad data at recovery time. These two uses require different types of taint. Thus, besides the clean/untainted tag, we distinguish *three* types of taint in DiskDuster, corresponding to three sources of taint:

**Untrusted ($\bar{\mathsf{U}}$).** We assign $\bar{\mathsf{U}}$ tags to all data from untrusted sources (like the network).
**Malicious ($\bar{\mathsf{M}}$).** We assign $\bar{\mathsf{M}}$ tags to all bytes written by malicious processes.
**Suspicious ($\bar{\mathsf{S}}$).** When a benign process reads $\bar{\mathsf{M}}$ bytes, we propagate that tag through the execution (see above). Thus, all writes of $\bar{\mathsf{M}}$ bytes to disk are also tagged $\bar{\mathsf{M}}$. However, even if the written data does not derive directly from $\bar{\mathsf{M}}$ data, it may have been influenced by it via an implicit flow. Thus, after a benign process reads $\bar{\mathsf{M}}$ or $\bar{\mathsf{S}}$ bytes, we label all its writes not already tagged $\bar{\mathsf{M}}$ with the $\bar{\mathsf{S}}$ tag.

*Monitor modules.* The two monitor modules trace both process execution and disk input/output. Specifically, the **disk monitor** keeps track of all reads and writes to disk, while the **process monitor** tracks running processes. When DiskDuster detects an attack, it marks the compromised process as 'malicious' and notifies the disk monitor. From now on, all writes by this process receive an $\bar{\mathsf{M}}$ tag. In the process monitor, malicious processes are handled in a special manner. For instance, when they start a new process, the created process will be marked malicious also.

*Logging, snapshotting and recovery.* The **logger** stores all information generated by the two monitor modules. The logs always include each write and read on disk, and in the case of a compromised process, they also contain detailed information about the context of the process. The **snapshot** module takes snapshots of the disk drive according to user-specified policies. Snapshotting allows us to skip backwards and forwards through an execution trace quickly. The **recovery** module, finally, sanitizes the system by replaying write operations from the last snapshot until the moment of infection.

### 3.3 Attack Detection

In our current prototype, DiskDuster detects attacks in one of two ways. First, it detects memory corruption and code injection attacks by means of dynamic taint analysis. The process is similar to other full system taint trackers like Argos [23] and Minos [6]. All data arriving from the network is marked 'untrusted' ($\bar{\mathsf{U}}$). Whenever such data modifies a process' control flow (e.g., when it ends up in the program counter), the process monitor treats it as an attack.

Second, when an external AV scanner detects new malware, we explicitly contact the process monitor to mark the corresponding process as malicious. The AV scanner is useful for attacks that do not compromise an existing program. For instance, a trojan installed by the user. Other detection methods can be plugged in easily. Regardless of how we detect the attack, from that point onwards, the process monitor tracks the malicious process.

### 3.4 The Process Monitor: Tracking Attacks at Thread Granularity

Upon detecting an attack, the process monitor closely monitors the offending process(es) to track which files and processes it influences and how. In the process, the process monitor classifies threads and processes as malicious, suspicious or benign. First, we explain these categories, and then we focus on technical challenges to support them. By default, all processes and threads are *benign* and the only exceptions are the malicious and suspicuous threads listed below.

*Malicious threads.* DiskDuster marks all processes corresponding to attacks reported by the AV scanner or DTA module as *malicious*. DiskDuster also treats a thread as malicious if it is attacked by local processes—for instance, when it uses a DLL provided by a malicious process, or when its parent is malicious. Once a thread has become malicious, all its writes are labeled with the $\bar{\mathsf{M}}$ tag. We say that a process is malicious if it has a malicious thread.

Thus, the process monitor should both identify the malicious thread, and inspect its execution context, such as the loaded dynamic libraries. Additionally, it tracks the creation of new processes by malicious threads and marks them malicious also.

*Suspicious threads.* As discussed in Section 2, accurate tracking of implicit dependencies is difficult, if not impossible. However, ignoring them causes false negatives. We take a conservative approach, and track *suspicious* threads—threads *possibly* influenced by malware—and ask users to verify the contents of suspicious files during recovery.

A benign thread becomes suspicious when we can no longer guarantee that malware does not influence its actions. First, whenever a benign thread has read a suspicious or malicious byte using I/O routines (e.g., from a file, the registry, or through interprocess communication), we consider it suspicious. Second, when a process has a malicious thread, we cannot rule out implicit flows between the malicious and benign threads. DiskDuster therefore considers all benign threads in this process suspicious. Finally, a child of a suspicious thread is also suspicious. We label all ostensibly clean data written by suspicious threads with the $\bar{\mathsf{S}}$ tag. We call any process with suspicious threads suspicious also. Thus, the process monitor again collects all information necessary to

identify a suspicious thread, and tracks the creation of its children. It also monitors the data passed through the I/O routines (e.g., file reads and writes).

**Low Level Tracking to Classify Processes.** Since the monitor resides at the (emulated) hardware level, process tracking is not trivial—normal process semantics as defined by the operating system are not readily available. The problem of extracting high-level semantic information from low-level data sources is known as the semantic gap, and has sparked much research activity in recent years [7,21]. We now discuss how DiskDuster bridges it.

*Process and thread identification.* To identify threads and processes at the level of a processor emulator, we use the solution proposed in Antfarm [12]. It tracks changes of the `cr3` (or page directory base) register, which stores the physical address of the page directory. As a rule, a context switch implies changing the set of active page tables, and thus loading `cr3` with the value stored in the descriptor of the process to be executed. DiskDuster uses `cr3` as a unique process identifier.

However, since all threads of a process share the page table directory, this mechanism does not distinguish between threads. To increase the granularity of tracking, the process monitor additionally looks up kernel-level data structures that hold process information. In 32-bit Windows, the Thread Environment Block (TEB), pointed to by the `FS` register, stores information about the currently running thread. As DiskDuster can easily reach this data structure from the emulator using the register, we extract all relevant thread information directly from the TEB.

*Tracking semantics.* In 32-bit Windows, the process monitor tracks the necessary semantic information by intercepting a number of functions from the `kernel32` library. These include the process creation functions, and the I/O routines, such as the file and registry read functions, or the interprocess communication functions. To determine addresses of these functions, the process monitor implements a solution typically used by shellcode. Using the TEB, DiskDuster identifies first the Process Execution Block (PEB), and then the loaded modules. Each loaded module contains the addresses and symbol names of available functions. DiskDuster uses this information during calls, jumps, and returns, and checks (at the level of the emulated hardware) if the program counter indicates the entry point of a function we intercept. If so, it calls a registered hook.

### 3.5 The Disk Monitor

As illustrated in Fig. 3, the disk monitor tracks all reads and writes to disk to support two of DiskDuster's main tasks: (1) restore the disk drive to a pre-infection state, (2) for all post-infection disk activity, present the user with an analysis of clean and suspicious files (so that she can safely keep the clean ones, and verify the suspicious ones).

The first task requires a replay of all disk writes that took place in the period between the last uncorrupted snapshot and the attack. The disk monitor simply logs all operations which modify data on disk, so that they can be repeated later.

Since the analysis phase requires precise information about clean, suspicious, and malicious parts of the disk, DiskDuster extends its taint tracking module to handle disk

operations, and stores taint values of the disk contents in a disk shadow map. Whenever a process stores data to disk, the disk monitor checks whether it should label these bytes with a tag. If the process is listed as suspicious or malicious, the data is labelled with $\bar{\mathsf{S}}$ or $\bar{\mathsf{M}}$, respectively. Similarly, if the bytes carry a $\bar{\mathsf{U}}$, $\bar{\mathsf{S}}$, or $\bar{\mathsf{M}}$ tag already, DiskDuster simply propagates it to the disk map. Conversely, when the program reads data from disk, the disk monitor propagates tags from the disk map into the main memory map. For instance, when a program reads tainted bytes from disk into memory, DiskDuster tags the corresponding bytes with a tainted tag in the memory map.

The diskmap can store the disk taint information at block level or at byte level, depending on the user's needs. The block level would provide information about which files were touched by an attack, while the byte level would be more specific and show which exact bytes in the files were changed by the malicious process. In the evaluation we used a byte level map. The taint propagation between the disk map and the main memory map is done at the level of the IDE emulator of the VM.

### 3.6 Snapshots

Once DiskDuster detects an attack, it reverts the disk to a pre-infection state by replaying disk writes that took place before the infection. Since replaying the execution from boot time would incur a high overhead, DiskDuster uses disk snapshots. Upon detecting an attack, it searches for the last snapshot before the infection, and replays only the disk writes that happened since. DiskDuster's snapshots are subject to simple policies, like "snapshot at fixed time intervals", or "snapshot after $n$ disk writes". For our experiments, we use the second option, and snapshot when the total number of writes equals 10% of the disk size[1]. In practice, this occurred approximately every 10 hours.

Suspending the execution may lead to undesired consequences, such as time-outs on network connections. To avoid such problems, DiskDuster implements live snapshots of disk drives. Once it triggers a snapshot, DiskDuster creates a copy of the drive in the background while the VM keeps running. During this process, all writes to disk generate a copy of the modified blocks to the snapshot before they are committed to disk.

### 3.7 Recovery and Analysis

System recovery begins when the user has shut down the machine. As illustrated in Fig. 3, DiskDuster starts by reverting the disk to a pre-infection state, then closely monitors infected processes to find out which files, folders and processes they influenced.

First, DiskDuster determines the initial intrusion moment, or more specifically, the first disk write by a malicious process. In the case of an attack detected by DTA, the intrusion moment is fixed exactly at the point where a good process turns into a bad one. If an AV software detects the infection, DiskDuster scans the logs for an operation that modifies a disk block corresponding to a file matching the AV signature. In either case, DiskDuster replays disk operations which took place before the first malicious write, and it provides the user with a disk in a benign state.

[1] The blocks need not be unique, so we snapshot also if the same 2GB of a 200GB drive are written 10 times.

Next, DiskDuster monitors offensive processes to log the data they influence (Sections 3.4-3.5), and presents a user with a list of clean, and suspicious ($\bar{S}$) files (malicious data is removed automatically). To map clean and suspicious disk blocks to file names, we use an ntfs library [25] to read the filesystem metadata from the physical disk. We extract the semantics of the filesystem to find the file name corresponding to a block.

As the disk monitor works at the physical level, a block on disk can be located in one of the following regions: (a) inside the filesystem and belonging to the data runs [2] of a file, (b) inside the filesystem but in a free region (i.e. not used by any file), (c) in the filesystem's metadata, or (d) outside the filesystem (e.g., the region of physical sectors 0-63 used by the bootloader). The reason for tracking writes outside the file system regions, is that these sectors are used by advanced malware like TDL4.

The DiskDuster resolver provides a list of filenames for the blocks that belong to filesystem objects like files, folders or metadata, and keeps additional information (like region mappings) for the other blocks. Normal users will be interested mainly in the file names, but security professionals may be interested also in the other information.

## 4 Evaluation

We now evaluate the effectiveness of DiskDuster in recovering from attacks. We do not focus on performance, other than to say that the slowdown of 20x during analysis on the replay side is no worse than that of other full-system DTA solutions [4,23,6]. Moreover, the overhead is sufficiently small to make the replay side keep up (in fact, previous experiments in decoupled security show that even with DTA slowdowns of 100x, the replayer keeps up with the production system, because of the idle time boost [4,10]).

In the remainder of this section, we use DiskDuster to recover from a variety of attacks.

### 4.1 Experimental Setup

We ran DiskDuster on a machine with an Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz, with 6MB cache, 4GB of RAM memory, and a SATA disk drive. The operating system running on the host was Linux with kernel version 2.6.32. As the guest we ran Windows XP with RAM memory size of 1GB, and a disk drive of 3GB with an NTFS partition stored in the *raw* format.

### 4.2 Workloads

To evaluate DiskDuster, we observe how well it recovers from an attack which has happened at a point in the past. We assume that malware is active for a while, and observe how much data modified by the user in the time between the infection and the attack DiskDuster can restore. To test with workloads that are both realistic and repeatable, we recorded several real Windows XP sessions using ReMouse[3], and replayed them once

[2] File content is made of data runs—lists of disk blocks with the actual content of the file

[3] `www.remouse.com`

the machine has been infected. The workloads contain the active use of a variety of applications, including the Internet Explorer 6.0 web browser, the FoxIT PDF reader, the standard Windows Picture and Fax Viewer photo editor, and the Notepad++[4] source code editor. For the experiments in Sections 4.3-4.4, we use five workloads, four short ones (denoted `WS-x`), and one long one (denoted `WL-1`). The short ones are one hour each with different activities with detailed descriptions (see below), while the long one captures three working days of a researcher in our lab.

- `WS-1` - the user visits a number of webpages using IE 6.0 and stores the content of several of them, only to reload them from disk later.
- `WS-2` - using the FoxIT PDF reader, the user loads and reads several PDF documents.
- `WS-3` - the user writes and sends an email, downloads several pictures from the web (IE 6.0) and edits them with the Windows Picture and Fax Viewer photo editor.
- `WS-4` - in this session, the user writes a program using the Notepad++ source code editor and makes a drawing using MS Paint. In both cases, the user stores, reloads, modifies and saves the work in several files.
- `WL-1` - in this session, the user engages in a wide variety of activities corresponding to three full days of work.

### 4.3 Single Step Attacks

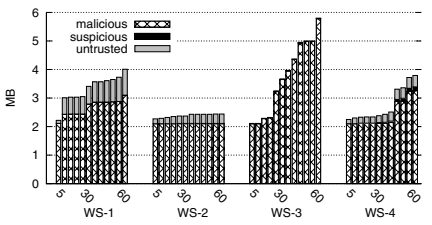
We first run DiskDuster with a set of straightforward attacks that do one or two things only—to verify that it can recover from malicious actions in isolation. For this purpose, we compromised the system using a drive-by-download from Metasploit (version 3.8.0-dev) and ran the following test attacks at the start of the short workloads: `WS-1`,..., `WS-4` (in each case, we "detect" the compromised process after exactly one hour):

(A) **Binary patch.** The malware binary downloaded modifies the executable file of a benign application (in this case, the IE 6.0 web browser, the FoxIT PDF reader, and MS Paint binaries). DiskDuster performs the analysis, and reverts the binary to its state before the attack.
(B) **Persistent drive-by download.** This time the malware adds a registry key to make itself persistent across reboots. DiskDuster performs the analysis, removes the binary, and restores the registry to its state before the attack.
(C) **File deletion.** The downloaded malware deletes a file from disk. DiskDuster performs the analysis, removes the binary, and reverts the deletion operation.

To evaluate the effectiveness of DiskDuster, we perform two sets of measurements: *infection rates*, and *recovery results*. Infection rates illustrate how quickly taint spreads over the disk. We present the amounts of suspicious, malicious, and untrusted disk data over time. Recovery results show the status of the disk after DiskDuster performed the analysis. We discuss how many benign files and folders a user can safely keep, and how many suspicious ones she needs to scrutinize. We focus on user data, but present results for both `\Documents and Settings`, and `\WINDOWS`.

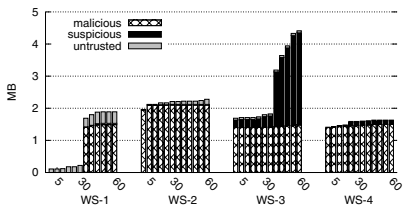
[4] `http://notepad-plus-plus.org/`

Fig. 5–7 show the result of these tests. The graphs present the spread of malicious, suspicious and untrusted data on the whole disk over time (at 5 min intervals), while the tables count the files containing malicious and suspicious bytes. The files are gathered into two categories: *need review*, and *temporary*. The user needs to scrutinize the former, while temporary files indicate data which she can flush, without any loss of work, for example, the cache and the `History` folder of IE 6.0, or the `dllcache` folder of the `\WINDOWS \system32` directory.



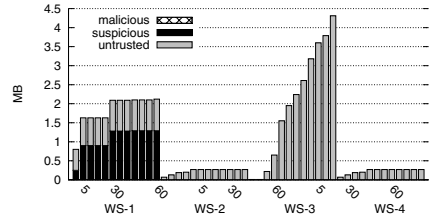

| WS | File info | Docum. and Settings | | | | WINDOWS | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Malicious | | Suspicious | | Malicious | | Suspicious | |
| | | Files | KB | Files | KB | Files | KB | Files | KB |
| 1 | Need review | 6 | 6.03 | 0 | 0 | 33 | 211.72 | 0 | 0 |
| 1 | Temporary | 5 | 19.11 | 0 | 0 | 5 | 113.37 | 0 | 0 |
| 2 | Need review | 0 | 0 | 1 | 4.54 | 8 | 169.67 | 13 | 1.68 |
| 2 | Temporary | 3 | 4.37 | 4 | 1.64 | 4 | 109.85 | 7 | 0.61 |
| 3 | Need review | 13 | 782.23 | 0 | 0 | 38 | 203.17 | 0 | 0 |
| 3 | Temporary | 23 | 120.07 | 0 | 0 | 5 | 118.80 | 0 | 0 |
| 4 | Need review | 11 | 374.68 | 2 | 250.74 | 27 | 193.35 | 8 | 35.87 |
| 4 | Temporary | 4 | 3.82 | 5 | 18.81 | 6 | 110.21 | 1 | 0.46 |

**Fig. 5.** The binary patch attack: infection rates and recovery results for four 60 minute workloads




| WS | File info | Docum. and Settings | | | | WINDOWS | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Malicious | | Suspicious | | Malicious | | Suspicious | |
| | | Files | KB | Files | KB | Files | KB | Files | KB |
| 1 | Need review | 1 | 98.85 | 1 | 3.83 | 21 | 736.85 | 37 | 62.61 |
| 1 | Temporary | 8 | 167.34 | 116 | 789.45 | 1 | 0.03 | 2 | 8.96 |
| 2 | Need review | 0 | 0 | 1 | 10.14 | 7 | 168.62 | 21 | 6.55 |
| 2 | Temporary | 2 | 2.51 | 5 | 3.00 | 2 | 0.18 | 1 | 0.18 |
| 3 | Need review | 3 | 102.27 | 12 | 1183.06 | 25 | 1440.36 | 10 | 56.02 |
| 3 | Temporary | 18 | 10.34 | 8 | 102.67 | 1 | 2.17 | 1 | 3.43 |
| 4 | Need review | 3 | 242.12 | 7 | 377.20 | 15 | 727.69 | 6 | 7.50 |
| 4 | Temporary | 7 | 9.03 | 3 | 27.08 | 0 | 0 | 0 | 0 |

**Fig. 6.** The drive-by-download attack: infection rates and recovery results for four 60 minute workloads




| WS | File info | Docum. and Settings | | | | WINDOWS | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Malicious | | Suspicious | | Malicious | | Suspicious | |
| | | Files | KB | Files | KB | Files | KB | Files | KB |
| 1 | Need review | 4 | 1.47 | 0 | 0 | 32 | 46.33 | 7 | 39.00 |
| 1 | Temporary | 71 | 452.20 | 1 | 44.56 | 1 | 3.50 | 0 | 0 |
| 2 | Need review | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | Temporary | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | Need review | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 3 | Temporary | 0 | 0 | 0 | 0 | 0.02 | 0 | 0 | 0 |
| 4 | Need review | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | Temporary | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 7.** The file deletion attack: infection rates and recovery results for four 60 minute workloads

We make a few observations. First, `WS-3` is more aggressive in spreading suspicious and untrusted bytes. This makes sense, as the user downloads a fair amount of data and then edits it. All these bytes are at least untrusted, and if the browser was malicious, then all the edits and subsequent writes of these benign processes are suspicious. Second, the three attacks have very different profiles in the way they spread malicious, suspicious and untrusted bytes. This makes sense also, as some attacks make multiple applications malicious, and thus spread more malicious bytes (e.g., the binary patch),

while others do not contain much malicious data at all (e.g., the file delete). Finally, we see that the number of files left malicious or suspicious is small—typically, these are files downloaded by a malicious process and processed by another process. Most of the user data was recovered.

### 4.4 Complicated, Real-World Attacks

In this section, we use DiskDuster to recover from four complex attacks involving real world malware, including the Win32/Sality virus [29], the Win32/Alureon trojan [17] and the Win32/Hupigon backdoor [16]. We follow the advanced attack scenario of Fig. 2, and test four malicious binaries in step 4. After launching an attack, we replay the long workload `WL-1`, which captures three working days (Section 4.2). We again detect the attack at the end of the workload, prompting DiskDuster to start its analysis.

In each experiment, the user first infects IE 6.0 by visiting a malicious website. We use Metasploit's meterpreter to migrate the attack from the browser to another application (e.g., the calculator). It deletes the antivirus program, downloads new malware to disk, and executes it. Apart from its normal malicious activities, the malware adds a registry key to make itself persistent across reboots, encrypts the Documents folder on disk, for ransom purposes, and deletes itself from disk[5]. In all cases, DiskDuster was able to restore the disk, undo the encryption, recover the AV scanner, etc.

In the remaining part of this section, we discuss the tested attacks in detail.

(I) **Hupigon backdoor.** Win32/Hupigon [16] is a backdoor, which provides an attacker with access to, and control of, an infected machine. Hupigon registers its component as a service.

(II) **Sality virus.** Win32/Sality [29] infects executable files. It replaces the original host code at the entry point of the executable to redirect execution to the polymorphic viral code, which has been encrypted and inserted in the last section of the host file. In addition, W32/Sality searches for specific registry subkeys to infect the executable files that run when Windows starts.

(III) **Alureon trojan.** Win32/Alureon [17] is a trojan that allows attackers intercept Internet traffic in order to gather confidential information such as user names, passwords, and credit card data. It may also allow to transmit malicious data to the infected computer.

(IV) **Zhelatin email worm / rootkit.** Zhelatin [9] spreads in e-mails with war-related subjects as an attachment named "video.exe", "movie.exe", "click me.exe" and so on. After start-up, it drops a randomly named file into the same folder where it was started from and runs it; this file installs a rootkit and p2p (peer-to-peer) component into the Windows System folder. In addition, it kills processes corresponding to virus scanners.

Fig. 8-13 show the results of these tests. The graphs present the spread of tainted data on the whole disk over time. The tables count the files containing malicious and suspicious bytes for two attacks which perform lots of system activities: the Win32/Sality virus

[5] In record/replay, AV scanners can still detect it, as the full execution trace is available.

and the Win32/Zhelatin email worm/rootkit. Observe that similarly to Section 4.3, taint spreads quite aggressively, and it is again expected. For example, all files a malicious IE 6.0 process stores in the `Temporary Internet Files` folder, become malicious as well. Next, since DiskDuster reverts the disk to a pre-infection state (while keeping most recent changes in the user directory), we are not so concerned about the taint in the system files. Finally, observe that the number of other files is small—these are again typically files downloaded by a malicious process, and modified by another one.

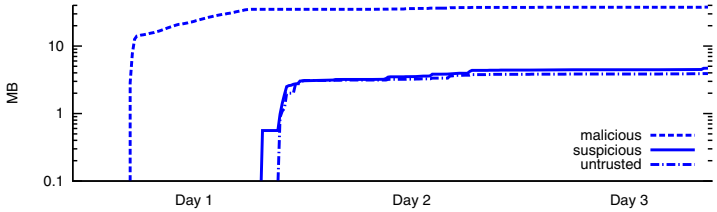



**Fig. 8.** The Win/32 Hupigon backdoor: infection rates for the `WL-1` workload

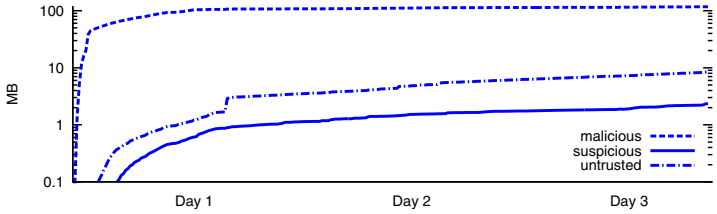



**Fig. 9.** The Win/32 Sality virus: infection rates for the `WL-1` workload

| **Documents and Settings\diskduster** | | | | | **WINDOWS** | | | | |
|---|---|---|---|---|---|---|---|---|---|
| File info | Malicious | | Suspicious | | File info | Malicious | | Suspicious | |
| | Files | KB | Files | KB | | Files | KB | Files | KB |
| **Need review** | | | | | | | | | |
| My Documents | 11 | 25253.30 | 0 | 0 | repair | 2 | 14.33 | 1 | 0.21 |
| Local Settings \Application Data | 3 | 7.18 | 0 | 0 | Microsoft.NET | 24 | 104.66 | 1 | 3.21 |
| Documents and Settings | 9 | 950.50 | 1 | 53.98 | system32 | 53 | 927.13 | 7 | 16.30 |
| - | - | - | - | - | WINDOWS | 32 | 173.08 | 6 | 113.24 |
| **Temporary files** | | | | | | | | | |
| Local Settings \Temporary Internet Files | 65 | 566.79 | 1 | 16 | system32 \dllcache | 14 | 87.18 | 1 | 1.19 |
| Local Settings \Temp | 34 | 74.75 | 2 | 6.52 | - | - | - | - | - |
| Local Settings \History | 3 | 23.85 | 1 | 0.62 | - | - | - | - | - |
| Recent | 5 | 4.16 | 1 | 1.19 | - | - | - | - | - |

**Fig. 10.** The Win/32 Sality virus: recovery results for the `WL-1` workload. L S = Local Settings; T I F = Temporary Internet Files

## 5 Limitations

As DiskDuster automatically recovers in the majority of cases and for very complicated attacks, a valid question is: why not in all cases, and why do we recover user data only—rather than the full system state? The answer is that there are subtle scenarios that are problematic or impossible for DiskDuster to solve. They are related to implicit flows.

The first problematic scenario concerns implicit flow in the user's head. We already discussed it in Section 2: a user makes a note (in a memo, say) about the absence of an AV scanner. As the information flows via the user's mind, DiskDuster cannot detect it.

The second problematic scenario concerns checksums on data structures with malicious data. While not too common, the system occasionally performs a calculation over data structures that contain malicious ($\bar{M}$) data. For instance, consider an OS-level linked list with a checksum. Both malicious processes and benign applications add nodes to the list and update the checksum. Whenever DiskDuster detects malware, the recovery process removes all malicious nodes from the list. However, doing so corrupts both the list and its corresponding checksum. The correct action would be to remove the malicious nodes and all nodes dependent on the malicious nodes, and then to restore the checksum. This is not possible without detailed semantic knowledge about the list. Unfortunately, since the OS sometimes stores such data structures on disk, we may end up with a corrupt system.

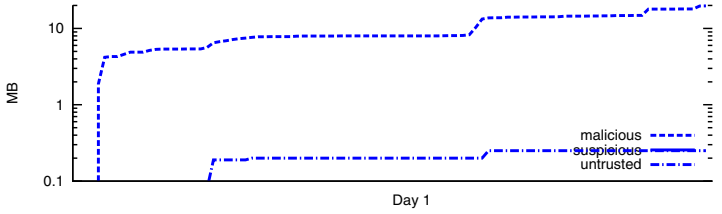



**Fig. 11.** The Win/32 Alureon trojan: infection rates for the `WL-1` workload. (Due to some problems with the replaying software, we limit the results to one working day.)

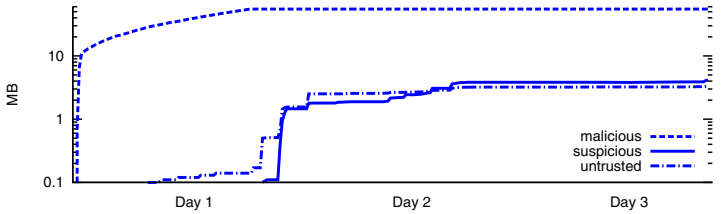



**Fig. 12.** The Win/32 Zhelatin email worm/rootkit: infection rates for the `WL-1` workload

| **Documents and Settings\diskduster** | | | | | **WINDOWS** | | | | |
|---|---|---|---|---|---|---|---|---|---|
| File info | Malicious | | Suspicious | | File info | Malicious | | Suspicious | |
| | Files | KB | Files | KB | | Files | KB | Files | KB |
| **Need review** | | | | | | | | | |
| My Documents | 1 | 17.50 | 1 | 4.80 | repair | 0 | 0 | 2 | 8.75 |
| UserInfo | 0 | 0 | 1 | 32.06 | Microsoft .NET | 3 | 28.00 | 15 | 57.85 |
| Cookies | 0 | 0 | 2 | 72.95 | system32 | 23 | 1283.74 | 13 | 43.07 |
| NTUSER.DAT | 1 | 1.20 | 0 | 0 | WINDOWS | 16 | 122.55 | 20 | 105.78 |
| **Temporary files** | | | | | | | | | |
| Local Settings\Temporary Internet Files | 0 | 0 | 50 | 702.98 | system32\dllcache | 9 | 77.00 | 2 | 0.69 |
| Local Settings\History | 0 | 0 | 2 | 13.99 | - | - | - | - | - |
| Local Settings\Application Data | 0 | 0 | 1 | 1.59 | - | - | - | - | - |
| Recent | 0 | 0 | 5 | 3.88 | - | - | - | - | - |

**Fig. 13.** The Win/32 Zhelatin email worm/rootkit: recovery results for the `WL-1` workload

To ensure correctness in the presence of implicit flows, DiskDuster currently restores the entire file to an benign version if any part of the file is flagged malicious. This results in correct recovery, but drops more benign writes than strictly necessary.

Finally, there may be implicit dependencies on restored files. Consider again a linked list manipulated by both malicious and benign processes and a benign process that reads a few benign nodes from the list and writes them out to a log. As it does not read $\bar{M}$ or $\bar{S}$ data, it remains benign throughout its lifetime. At some point, DiskDuster restores the file with the linked list to a previous safe state, as explained above.

The problem is: what do we do with the benign process' log file? Because of the implicit dependency on the file (and its malicious contents), we cannot keep it as is, lest we introduce inconsistencies. Thus, we track the fact that the read accessed a file that DiskDuster restored, thus making the log file a candidate for restoration also. And so on. The additional roll-backs keep the system consistent, but again lead to possibly dropping a few more benign writes than strictly needed.

## 6 Related Work

*Decoupled security checks.* Recording and replaying is used in many research projects [8,1,19]. Full decoupled security for virtual machines was introduced by VMware [4], and the model was quickly picked up byothers (e.g., for mobile phones [24] and fast Xen VMs [33]). AfterSight [4] comes closest in spirit to the record and replay side of DiskDuster. However, all these systems differ from DiskDuster in that they limit themselves to attack detection and leave remediation to the administrator.

*Data recovery.* Most automated attack recovery systems either focus solely on data on disks (much like advanced versioning systems), or rely on the support of the target OS—either in the form of a module inside the victim's machine [15,5,11,28,32,20,14,3], or a proxy [34,28].

Many of these projects depend on external methods to indicate the root cause of an infection, and to obtain high level semantics (e.g., the way in which the OS uses the password file, the dependencies between OS-level operations, etc.). Such information facilitates the process of intrusion recovery, and aids in building dependency graphs [15,11,32,14], and behavior models [20]. As a result, the analysis becomes more detailed than in systems which operate at the machine level, like DiskDuster.

However, since we cannot assume the integrity of the kernel of a monitored system, it is possible that attacks hinder the analysis, for example by modifying the logs or the dependency graphs. In contrast, DiskDuster carries out a comprehensive analysis without relying on any kernel support whatsoever, and is still able to recover from very sophisticated attacks. We now discuss the most related projects in more detail.

In Wayback [5] versioning is automatic at the write level: each write to the file creates a new version, so that access to any previous version is possible. Wayback needs knowledge about the filesystem and modifies the monitored system. Similarly, BackTracker [15] is implemented inside the OS and tracks OS objects. It extracts dependencies between different components as the attack evolves and can produce dependency graphs.

Taser [11] uses a kernel module to log kernel operations on processes, filesystems and the network for Linux systems. The analysis is decoupled and assumes that the kernel of the monitored host is not compromised. Using the semantic information it constructs detailed dependency graphs to track data flows.

SEE [28] explores one way isolation for Linux processes—processes do not share the disk, and all their commits are written in different locations. It achieves such isolation by interpositioning at the level of system calls and the virtual filesystem layer, using copy on write implemented as file copy operations. Essentially, it is a filesystem proxy implemented as kernel modules that creates a shadow drive for the process. At the end of execution, it either commits or discards the changes based on user input. Thus, the user must review *all* changes. With DiskDuster users review only suspicious data, while DiskDuster restores the malicious bytes.

Paleari et al. [20] aim to generate remediation procedures to purge infections from a system, but the system can only recover system state and some system files of Windows, and cannot handle deleted files. The system records system calls executed in the emulated environment and infers behavior models based on sequences of the system calls and their parameters.

Retro [14] is a recovery system for Linux that relies on a kernel module to generate action history graphs. The design assumes that the kernel, filesystem, checkpoints or logs are always safe. Another crucial assumption is that the infection is discovered very quickly, otherwise the graphs become too hard to manage. After detecting an infection, the system reexecutes processes and may block if user input is needed and wait for the input in order to continue. In contrast, DiskDuster successfully recovers from attacks that have been active for days.

Back to the Future [3] removes malware and helps users repair systems after an attack. The implementation is Windows specific and requires significant user interaction. The user needs to define *a priori* which is the trusted data and only modifications of this data are logged. Moreover, the user has to decide what to do whenever an untrusted program interferes with a trusted program. The framework is selective about the monitored system calls and may also decide to terminate a process and inform the user.

## 7 Conclusion

We have described DiskDuster, an attack analysis and recovery system capable of removing all traces from complicated attacks. DiskDuster relies on execution trace recording, snapshotting, and especially taint analysis to track a malcode's actions. Although an attack may be detected long after the infection, DiskDuster is able to roll back to the initial point of infection and restore the disk to that state.We demonstrated the power of our system with complicated and real-world attacks.

DiskDuster greatly helps the analysis of an attack by the classification of bytes located on the physical drive into trusted, malicious and suspicious(which may be the result of implicit flows). Using DiskDuster, the user can recover all post-attack data which was not touched by the attack and is still clean.

**Acknowledgments.** This work is supported by the EU through project ERC-2010- StG 259108-ROSETTA, DG Home iCode and FP7-ICT-257007 SYSSEC.

## References


1. Basrai, M., Chen, P.M.: Cooperative Revirt: Adapting message logging for intrusion analysis. Technical Report CSE-TR-504-04, University of Michigan (2004)
2. Cavallaro, L., Saxena, P., Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 143–163. Springer, Heidelberg (2008)
3. Chen, H., Hsu, F., Li, J., Ristenpart, T., Su, Z.: Back to the future: A framework for automatic malware removal and system repair. In: Proc. of CCS (2006)
4. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling dynamic program analysis from execution in virtual environments. In: USENIX ATC (June 2008)
5. Cornell, B., Dinda, P.A., Bustamante, F.E.: Wayback: A user-level versioning file system for Linux. In: Proceedings of USENIX 2004 (Freenix Track) (2004)
6. Crandall, J., Chong, F.: Minos: Control data attack prevention orthogonal to memory model. In: 37th International Symposium on Microarchitecture (2004)
7. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: Narrowing the semantic gap in virtual machine introspection. In: S&P (2011)
8. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In: Proc. of the Symposium on Operating Systems Design and Implementation, OSDI (2002)
9. F-Secure: Email-Worm:W32/Zhelatin.CQ, `http://www.f-secure.com/v-descs/email-worm_w32_zhelatin_cq.shtml`
10. Folkerts, A., Portokalidis, G., Bos, H.: Multi-tier Intrusion detection by means of replayable virtual machines. Technical Report IR-CS-47, VU University (2008)
11. Goel, A., Po, K., Farhadi, K., Li, Z., de Lara, E.: The taser intrusion recovery system. SIGOPS Oper. Syst. Rev. 39, 163–176 (2005)
12. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: tracking processes in a virtual machine environment. In: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference (2006)
13. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium, NDSS 2011 (2011)
14. Kim, T., Wang, X., Zeldovich, N., Frans Kaashoek, M.: Intrusion recovery using selective re-execution. In: Proc. of OSDI 2010, Vancouver, Canada (2010)
15. King, S.T., Chen, P.M.: Backtracking intrusions. ACM Trans. Comput. Syst. 23(1), 51–76 (2005)
16. Microsoft Malware Protection Center: Backdoor:Win32/Hupigon, `http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?name=Backdoor%3AWin32%2FHupigon`
17. Microsoft Malware Protection Center: Trojan:Win32/Alureon.FE, `http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?name=Trojan:Win32/Alureon.FE`
18. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the 12th Annual Network and Distributed System Security Symposium, NDSS (2005)
19. Oliveira, D.A.S., Crandall, J.R., Wassermann, G., Felix, S., Zhendong, W., Frederic, S., Chong, T.: ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In: ASID 2006 (2006)
20. Paleari, R., Martignoni, L., Passerini, E., Davidson, D., Fredrikson, M., Giffin, J., Jha, S.: Automatic generation of remediation procedures for malware infections. In: Proceedings of the 19th USENIX Conference on Security (2010)

21. Pfoh, J., Schneider, C., Eckert, C.: Exploiting the x86 architecture to derive virtual machine state information. In: Proc. of SECURWARE 2010 (2010)
22. Porras, P., Saïdi, H., Yegneswaran, V.: A foray into conficker's logic and rendezvous points. In: Proc. of LEET 2009 (2009)
23. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks. In: ACM SIGOPS EuroSys 2006 (2006)
24. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Android: Versatile Protection for Smartphones. In: Proc. of ACSAC (2010)
25. The Linux-NTFS Project, http://www.linux-ntfs.org
26. Slowinska, A., Bos, H.: Pointless tainting? evaluating the practicality of pointer tainting. In: Proceedings of ACM SIGOPS EUROSYS (March-April 2009)
27. Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., Vigna, G.: Your botnet is my botnet: analysis of a botnet takeover. In: Proc. of CCS 2009, New York, NY, pp. 635–647 (2009)
28. Sun, W., Liang, Z., Sekar, R., Venkatakrishnan, V.N.: One-way isolation: An effective approach for realizing safe execution environments. In: Proc. of NDSS (2005)
29. Symantec: W32.sality, http://www.symantec.com/security_response/writeup.jsp?docid=2006-011714-3948-99
30. Verbowski, C., Kiciman, E., Kumar, A., Daniels, B., Lu, S., Lee, J., Wang, Y.M., Roussev, R.: Flight Data Recorder: Monitoring persistent-state interactions to improve systems management. In: 7th USENIX OSDI (2006)
31. VMWare. Vmware workstation 6.5 beta release notes (August 2008), http://www.vmware.com/products/beta/ws/releasenotes_ws65_beta.html
32. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007 (2007)
33. Zhang, S., Jia, X., Liu, P., Jing, J.: Cross-layer comprehensive intrusion harm analysis for production workload server systems. In: Proc. of ACSAC 2010 (2010)
34. Zhu, N., Chiueh, T.: Design, implementation, and evaluation of repairable file service. In: The International Conference on Dependable Systems and Networks (2003)

# NetGator: Malware Detection Using Program Interactive Challenges

Brian Schulte, Haris Andrianakis, Kun Sun, and Angelos Stavrou

The Center for Secure Information Systems,
George Mason University,
4400 University Drive, Fairfax 22030, USA
{bschulte,candrian,ksun3,astavrou}@gmu.edu
http://csis.gmu.edu

**Abstract.** Internet-borne threats have evolved from easy to detect denial of service attacks to zero-day exploits used for targeted exfiltration of data. Current intrusion detection systems cannot always keep-up with zero-day attacks and it is often the case that valuable data have already been communicated to an external party over an encrypted or plain text connection before the intrusion is detected.

In this paper, we present a scalable approach called *Network Interrogator (NetGator)* to detect network-based malware that attempts to exfiltrate data over open ports and protocols. NetGator operates as a transparent proxy using protocol analysis to first identify the declared client application using known network flow signatures.Then we craft packets that "challenge" the application by exercising functionality present in legitimate applications but too complex or intricate to be present in malware. When the application is unable to correctly solve and respond to the challenge, NetGator flags the flow as potential malware. Our approach is seamless and requires no interaction from the user and no changes on the commodity application software. NetGator introduces a minimal traffic latency (0.35 seconds on average) to normal network communication while it can expose a wide-range of existing malware threats.

## 1 Introduction

Targeted and sophisticated malware operates unhindered in the enterprise network. Indeed, the Anti-Phishing Working Group (APWG) [1] reported that the first six months of 2011, data-stealing malware and generic Trojans increased from 36% of malware detected in January, 2011 to more than 45% in April, 2011. Sophisticated malware utilizes obfuscation and polymorphic techniques that easily evade anti-virus and intrusion detection systems. A study by Cyveillance[11] showed that popular anti-virus solutions only detected on average less than 19% of zero day malware increasing only to 61.7% on the 30th day.

Once inside the host, malware can establish command and control channels with external points of control, often controlled by a single entity called a botmaster, and form drop points to exfiltrate data. To avoid detection, malware

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 164–183, 2013.

utilizes legitimate and usually unfiltered ports and protocols, including popular protocols such as HTTP, to establish these communications. Due to the volume of network traffic, enterprises are unable to effectively monitor outbound HTTP traffic and discern malware from legitimate clients. Current botnet detection systems focus on identifying the botnet lifecycle by looking for specific observables associated with either known botnets or typical botnet behaviors. These approaches suffer from the fact that malware continues to evolve in sophistication improving their ability to blend into common network behaviors.

Additionally, current systems are unable to inspect encrypted communications, such as HTTPS, leaving a major hole that malware will increasingly capitalize on. The use of encrypted traffic has been growing as web applications begin utilizing HTTPS for its privacy benefits. For example, Facebook recently announced HTTPS as an optional protocol for accessing its site. While an improvement for privacy, the use of HTTPS poses major technical hurdles for current network monitoring and malware detection.

In this paper, we demonstrate the results of a novel approach called *Network Interrogator (NetGator)* to detect and mitigate network-based malware that (ab)uses legitimate ports and protocols to initiate outbound connections. NetGator operates as a transparent proxy situated in the middle of all network communications between the internal clients and external servers. Our approach consists of two phases that are real-time and completely transparent to the user. In the first phase, we employ a passive traffic classification module that performs protocol analysis to first identify the advertised client application type (i.e. browser, program update, etc.). We do so based on existing network signatures for legitimate applications including the use of ordering of traffic headers that sometimes characterizes the host application. The main purpose of this first order flow classification is to determine the claimed or declared identity of the end-point software that generated the traffic into two classes: potentially legitimate or unknown.

As a second phase, for flows that we can successfully classify as part of potentially legitimate applications inside the organization (e.g., approved browser using HTTP(S) on port 80 or 443), Netgator attempts to further probe the endpoint application by inserting itself as part of the network communication.We do so by issuing a challenge back to the client that exercises existing functionality of the legitimate application. A challenge is a small, automatically generated piece of data in the form of an encapsulated puzzle that a legitimate application can execute and automatically respond without any human involvement. If the application is unable to correctly solve and respond to the challenge, NetGator will flag the source as potentially being malware and optionally sever the connection along with reporting the offending source. Therefore, rather than attempting to classify network traffic as either good or bad based on network (packet, flow, or content) inspection, the second phase of NetGator focuses on validating that the traffic stems from a legitimate application.

The proposed approach is an automated twist of the Human Interactive Proofs mechanism (e.g., CAPTCHAs), but focused on verifying program internal

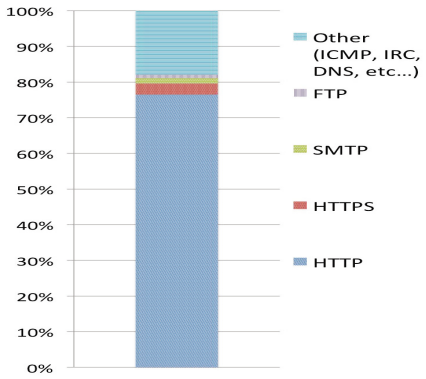



**Fig. 1.** Study of 1026 samples from popular classes of zero-day malware

functionality rather than humans. We call this approach Program Interactive Challenges (PICs). We define a PIC as a challenge comprised of a request and expected response pair which tests for existing functionality of legitimate applications. A PIC can be generated when there is an end-program state that has a deterministic programmable network response and that state can be triggered by communication with the server. The complexity of the PIC depends on the complexity required to implement that state on the client side. The intuition is that if the challenges are diverse enough and exercise complex functionality of the legitimate applications, malware will have to either implement said functionality making it large or attempt to create application hooks or use the legitimate application to "solve" the puzzle. In the former case, the malware code will increase dramatically since malware has to now implement a lot of unnecessary and complex functionality, for instance, a JavaScript parser. In the latter case, the malware will have to farm out the traffic to the corresponding legitimate local end-point application to solve the puzzle. In addition, the malware will have to insert itself after the puzzle exchange while suppressing traffic from the legitimate application. Our approach raises the bar because it forces malware to perform additional invasive operations that would not be required without our system. It is not enough for the malware to just link to Browser libraries that implement communications, the malware has to also take over the HTML, Javascript, and Flash rendering engines. Therefore, NetGator increases the attack complexity for the adversary without requiring any human involvement.

We tested our system on 1026 zero-day malware samples in Windows virtual machines. Our experiments show that the majority of malware uses popular, unfiltered network ports to connect to remote servers for various purposes. Figure 1 shows that nearly 80% of the malware samples used HTTP/S for outbound communications. Therefore, we focus on developing PICs for browsers to show the effectiveness of our method. We can leverage HTML, Flash, Javascript, and other common browser components to form challenges for browsers. However, our approach is more general and can be extended to generate and use PICs for other applications (e.g., VoIP, OS updates) through analyzing the functionality

supported by the application software agents. Second, most malware only includes minimal functionality to reduce its size and avoid being detected, so it cannot correctly respond to the challenges. For example, many malware scripts use the "*wget*" command to download malicious code from external servers without compromising the browsers in the OS. Such scripts do not know how to respond to the PICs for the browsers. If a large-size malware includes all the challenging functionality for a browser or compromises the browsers in the OS, it can defeat our solution; however, we increase the bar for attacks to succeed.

We implemented a prototype NetGator system that includes different PICs for browsers. For non-text/html data, NetGator issues challenges when it receives the request packets from the client, which we refer call *request challenges*; for text/html data, it issues challenges when it receives the response packets from the external servers, which we call *response challenges*. Compared to request challenges, response challenges can reduce the overhead that might be introduced when enacting the request challenge on each HTTP request and prevent a malicious agent from downloading an executable that is disguised as an HTML file. However, it may lower the security by allowing the malicious request to complete even if the software agent is detected as malicious later. The experimental results demonstrate the effectiveness of PICs in identifying malware that attempts to imitate the network connection of popular browsers. It introduces an average of 353 milliseconds end-to-end latency overhead using request challenges and 24 milliseconds using response challenges.

In summary, we make the following contributions:

- We designed a malware detection system that utilizes a two-pronged approach to identify malicious traffic. We first classify traffic using passive inspection. For the flows that correspond to potentially legitimate applications, we "challenge" the host application by automatically crafting Program Interactive Challenges (PICs) that exercise complex functionality present in the legitimate application.
- We demonstrate the feasibility of our approach using HTML, Flash, Javascript, and other common browser components to form challenges for browsers. PIC was able to expose a wide-range of malware threats operating inside Enterprise networks.
- Netgator can be used in practice: it does not incur significant communication overhead and it does not require any user interaction or changes on the commodity application software.

## 2 Background

### 2.1 HTTP Headers and MIME Types

HTTP request and response packets contain various header elements which encase pertinent information about the transmission. Requests are prefaced by various headers notifying the server of what the client expects to receive. Responses

are accompanied by headers as well informing the client of what is being transmitted. Each browser uses a distinct header ordering. The same browsers also have slight differences depending on which operating system they are running. Our passive inspection module uses these unique orderings to create signatures, which can be used by the active challenge module to identify browers and pick appropriate PICs to challenge the browser. Multi-purpose Internet Mail Extension(MIME) types describe the content type of the message being transmitted. The main general MIME types are application, audio, image, text, and video. Since the majority of most web pages have the MIME type text/HTML that can be challenged on the response instead of the request, we can reduce the overall network delay caused by the active challenges.

Internet Content Adaptation Protocol (ICAP) allows for the modification and adaptation of HTTP requests and responses. All the various elements of HTTP messages can be edited. Typical uses of the ICAP protocol include actions such virus scanning or content filtering. The protocol relies on an ICAP client that forwards traffic to an ICAP server, which is in charge of the adaptation of requests/responses. In our implementation, we use the open source proxy Squid [5] as our ICAP client and use the open source option Greasyspoon [2] as our ICAP server. Greasyspoon allows scripts written in various languages (Java in our system) to act on all incoming requests and responses.

## 3 Threat Model and System Architecture

### 3.1 Threat Model and Assumptions

We assume that a client machine in an enterprise network may be infected with malicious code, and malicious code needs to "call home" and establish a back channel communication with remote server(s). The malware does not form connections immediately upon execution, but waits for an indeterminate amount of time or a user event before initiating network connection. Moreover, we assert that a certain subset of browser components and capabilities are necessary to navigate the Internet and confine our challenges to these.

The sophistication of most current malware has not yet reached the level of implementing or imitating the entire HTML, Javascript, or Flash engines and software stacks within themselves. The code size of a sophisticated malware will increase dramatically in order to include the functionalities for responding all the known active challenges, and make it prone to being detected. We assume malware that has infected a host does not wish to access the full software stack of the legitimate application software (e.g. browsers) that natively reside on the system in order to remain stealthy and launch attacks quickly.

NetGator utilizes a network-level transparent proxy to identify and filter out unknown traffic. For the traffic that matches programs that have been approved for the organization, we automatically craft active challenges to probe and verify end-point applications. Figure 2 depicts NetGator's system architecture. In the network, all traffic to be inspected is routed to this proxy. If the application passes both the passive inspection and active challenge, the proxy permits the

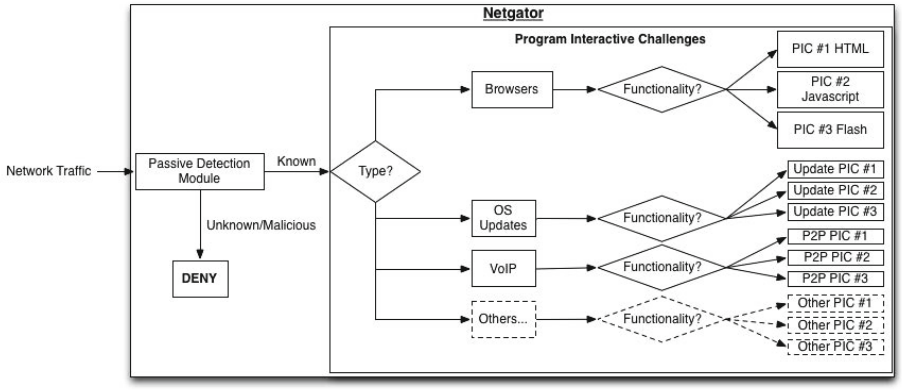



**Fig. 2.** NetGator System Architecture and Traffic Control Flow

outbound connection by forwarding the traffic to the default gateway. Connections that are either unknown or fail to pass the active test are dropped (or a human operator can be informed depending on the site's policy).

The network proxy consists of two major components: passive, signature-based flow inspection and active challenges. The passive inspection module acts as an initial filter, recognizing the known (and legitimate) category of the end-point software applications. We do not need the exact version of the end-point application but rathe its broader type (Browser, updater, etc.). Traffic from unknown applications can be treated preferencially allowing in the insertion of policies blocking, alerting, or even logging the flows that stem from such unmapped applications. This enables our system to adapt to new applications and network environments since new applications can be immediately recognized and mapped thus becoming "known". Traffic flows for which we already have a signature are issued active challenge(s) to further verify the legitimacy of the end-point software that generated the network traffic. The Program Interactive Challenges (PICs) are automatically generated by the proxy in advance and can chose from a wide variety of potential functionality based on the complexity of the end-point software. For instance, for browsers we show more than three different types of PICs that can be used.

To bootstrap in an enterprise network, traffic should be gathered for a period of time to decide what applications are operating on the network. Once armed with this information, we can utilize the passive inspection module to accomplish two tasks: filtering out malicious/unknown traffic and classifying known applications. First, by knowing which applications are expected to be traversing the network, we can drop any unknown, potentially malicious traffic as well as known malicious traffic. Second, the ability to classify the known applications allows us to send the corresponding active challenges to specific types of applications. We can derive passive signatures from packet header information collected from packet sniffers such as Wireshark [7]. These signatures are representative of the distinct HTTP header content and HTTP header ordering that each browser

possesses. However, the passive inspection module cannot provide timely filtering or blocking in zero-day attack situations. Moreover, since network requests can be easily altered, a request may be generated by malware masquerading as legitimate software. NetGator provides an active challenge mechanism to effectively detect and mitigate these attacks.

For a known application, the NetGator proxy can obtain the type and version of the supporting software from the passive signatures collected by the passive inspection. The proxy maintains a table that records the corresponding program interactive challenges (PICs) supported by each application software. Therefore, the proxy can send one or more PICs to the application that initiates the communication. For legitimate applications, they should be able to correctly respond to the challenges with their embedded functionalities. For example, we can leverage HTML, Flash, Javascript, and other common browser components to form challenges for browsers. If the application is unable to correctly solve and respond to the challenge, NetGator will flag the source as malicious.

Depending upon the type of requested data in the packet, the proxy can challenge either the request or the response. When challenging the network request the proxy receives a request from an application and blocks it, while returning a challenge as the response to the application's request. Only if the application can successfully solve the challenge and respond to the proxy, the proxy will forward the original request to the default network gateway. For instance, any HTTP request for application audio, or video data should be challenged and blocked until the challenge is solved. Challenging the network response allows the application's request to pass through but inserting our challenge into the original response from external servers and then sending it to the application. Legitimate applications can solve the challenges and notify the proxy. If the proxy cannot receive a correct answer from an application after sending the response challenge for a pre-defined time, it marks the software agent as malicious. For text/HTML requests, we can insert challenges into the response's text/HTML data. Challenging the request provides stronger security than challenging the response, since the application cannot obtain any useful data information from the external servers if it cannot pass the challenges. However, challenging the response can offer a smaller network latency, since the application only needs to solve the challenges after the response data has been received.

One of our primary design principle is to keep the system transparent to the end-user and adaptive to new software. Our approach does not require the user to prove or input anything, but shifts the onus of proof to the requesting application. Moreover, since our solution only utilizes the existing functionalities in commodity application software, we do not need to change their source code.

## 4 Design and Implementation

We design and implement both the passive inspection module and the active challenge module in NetGator, which supports both challenging at the request and challenging at the response. Since browsers represent a vast majority of

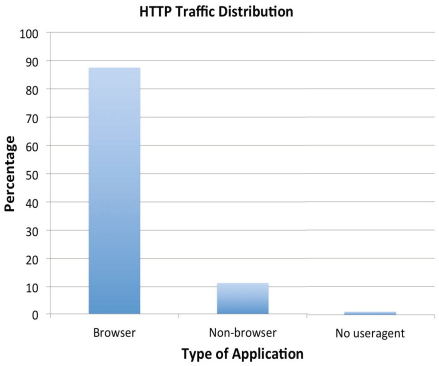



**Fig. 3.** Distribution of applications on port 80 of a large university network

traffic on a typical network and HTTP protocol is widely exploited by malicious code, we focus on developing PICs for validating browsers. However, our methods can easily be adopted to challenge other agents by identifying unique functionality that they possess including ones that also utilize HTTP/S.

### 4.1 Passive Inspection

The passive inspection module consists of two parts: signature generation and signature matching. First, it employs protocol analysis to first identify the advertised client application type based on its network communication signature. This signature is derived from a distinct ordering of traffic headers found in each client. Since different versions may support different sets of functions, it generates the signatures for different software agents with different version numbers and saves the signature in a data set. Second, it inspects the real time traffic, dynamically derives the signature of the user agent, and compares it with the signature seen to identify the user agent. The signature set is used to determine the claimed identity of the end-point software that generated the packets as known or unknown. Our passive module will drop the traffic from unknown client programs. Known traffic (e.g., http(s) through port 80 or 443), may pass through without being blocked or receive further inspection.

The signatures of user agents are generated automatically by observing traffic on the network and extracting HTTP header orderings. First a packet capture is performed on the network to gather information about which applications are running on the network. The pcap file is then exported into an XML file from Wireshark [7]. Once the data is in XML format, it is processed by a Python script. This script then extracts the message header and assigns a number to each HTTP header forming the signature. With unique signatures for each application that exists on the network we are able to issue the proper PIC(s) for each agent.

To perform real time detection, it sniffs the packets that traverse the network scanning for HTTP header instances that match our signature set of various user agents. A modified version of tcpflow [6], which we call *protoflow*, scans the

network traffic and pass the information to our identifier program. The change made is to write the data acquired by the capture to a space in memory which is shared by our second piece of software called *inspector*. Inspector takes the strings from the shared memory and compares the data against a collection of regular expressions that distinguish various user agents. These regular expressions are made up of the specific HTTP header ordering that are unique to each different browser.

We develop a string matching algorithm to check whether the "user-agent" string in the HTTP headers contains the name of browsers, including "Firefox", "Chrome", "Safari", "Opera", "MSIE 8.0", "MSIE 7.0", "MSIE 6.0". If yes, we label the packet as "browser", otherwise label it as "non-browser". For packets that do not provide any user-agent information, we label them as "non-labeled". We run the algorithm on the traffic of a large university for two hours, and label 8,825,177 packets as "browser", 1,143,040 packets as "non-browser", and 110,763 packets as "non-labeled". Figure 3 shows that traffic on port 80 (HTTP) is mostly comprised of browsers, but other applications can communicate via HTTP as well. The most common non-browser user agents include web-crawlers, application updates, bots, or spiders. For example, we observe 15,506 occurrences of web-crawlers. For those "non-browser" applications that utilize HTTP/S, we need to develop separate PICs for each type of user agents.

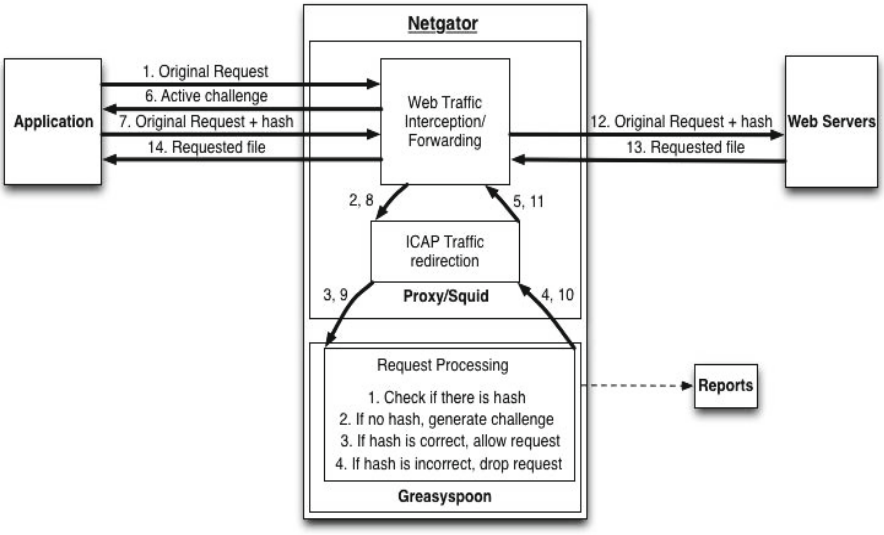



**Fig. 4.** Active Request Challenge Flow

### 4.2 Active Challenges

The active challenge module consists of a transparent proxy and an ICAP server. The proxy redirects all network request and response traffic on ports 80 and 443 to the ICAP server which then generates and verifies PICs to the client

software. Based on the Multipurpose Internet Mail Extensions (MIME) type of the requested data, the ICAP server either rewrites the response completely, or simply inserts additional code into the response code. If the response is any type of non-text/html data, the client request is blocked and a completely new response containing only our challenge is sent back to the client. For text/html types, the challenge code is inserted inside of the original response to reduce the network delay.

We use Squid 3.1.8 as the proxy for rerouting HTTP and HTTPS traffic and the ICAP server, Greasyspoon, for handling scripts on requests and responses. To handle HTTPS traffic, Squid's ssl-bump feature is harnessed. HTTPS traffic leaving the host is encrypted with a single key that the proxy possesses. Once the traffic reaches the proxy, it is decrypted and forwarded to Greasyspoon for generating and verifying challenges. Next, the proxy re-encrypts the request with the key established with the targeted external web server. This raises the concern that the client will never receive the external server's certificate, thus leaving it vulnerable to phishing sites. This can be handled by leveraging the trust of the Netgator proxy. That server can determine if a certificate that returns from an external site is legitimate or not. This method enables us to intercept HTTPS requests initiated by malware where other solutions would typically fail.

To keep track of the various connections, we leverage Greasyspoon's cache which contains a hashmap. For each IP address and user agent pair, this hashmap contain an entry that records whether the client passed a challenge, how many times it has been challenged, as well as how many times it passes the challenge. Thus, even if the malware correctly forms a user-agent string that is operational on the infected machine, the hashmap still can reflect that an entity on the system did not pass the challenge. The hashmap is periodically written to a log file available for inspection.

To reduce the number of challenges, the proxy automatically passes a network request for a page if the requesting client has passed a request for that page's domain. For example, if a client requests to access *www.foo.com/bar* and has already proven itself while requesting *www.foo.com*, the proxy will let it pass automatically. This enables us to lessen the burden from websites that trigger many GET requests for items such as images or flash objects. The list of "passed" domains is periodically cleared in order to catch malware that might use one of these sites for control communication. It is also possible to operate Netgator without keeping these records, and thereby issues challenges to each request, albeit with an increase in overhead.

**Challenging Network Requests.** When the proxy observes a request for non-text/html data, it issues a challenge to the client. The challenge can take various forms based on the functionality of the browser. Whichever test is administered, the driving element behind each of them is a redirect to the original requested URL with a hash appended to it. The only challenge that needs to contain more than a simple redirect command is the Flash challenge, which is a Flash object, not actual lines of code. The nature of the Flash challenge actually allows it to be more difficult for an attacker to bypass since the Flash object is embedded in the

```
<html>
<head>
<script type="text/javascript">
window.location = {URL requested}?=\
                  {hash generated}
</script>
</head>
<body></body>
</html>
```

(a) Javascript Challenge Code

```
<html>
<head>
<meta http-equiv='Refresh' content=\
     '0; url={URL requested}?=\
     {hash generated}'>
</head></html>
```

(b) HTML Challenge Code

```
…code to set up flash embedding…
<script type="text/javascript">
var params = {allowScriptAccess: "always"};
function GetURL(){return (URL requested);}
function GetHash(){return (hash generated);}
swfobject.embedSWF(\
        "http://(Gateway IP)/flashtest.swf",\
        …size parameters…, params);
</script>
```

(c) Flash Challenge Code

**Fig. 5.** Response with various active challenges

HTML page with Javascript, essentially combining two of our challenges into one. To correctly pass the requested URL and hash to the Flash object, Javascript functions are embedded and returned in the HTML code to interact with the object and provide it with the redirect information needed. If the client is able to correctly execute the challenge, Greasyspoon can match the new request with an appended hash to the originally requested URL. If the hash is correct, the request is allowed to pass; if the hash is incorrect, the request is dropped and an error message is sent to the client. Figure 4 shows the implementation for the request challenges.

The responsibility of Greasyspoon is to intercept the connection when it observes a request and send back a custom crafted response to the client in order to initiate the challenge. In order to correctly form the response, Greasyspoon executes a Javascript, which generates the hash and then crafts the new HTML code that will be returned to the client. A tuple of four factors is used to generate the hash: a static, secret key known only to the proxy, the requesting client's IP address, the URL being requested, and the current time's seconds value. The Javascript calculates a hash value from this tuple using SHA1. Next, this Javascript replaces the header and the body of the request in order to customize the response with the hash value for the client. The header must be replaced with a properly formed HTTP response header to signal Greasyspoon that a response is required to be sent back to the client directly from the proxy. For our implementation we use a standard HTTP/1.1 200 OK response HTML code

and insert a fragment of Javascript code that executes a redirect operation. The sample response codes using Javascript, HTMl, and Flash are shown in Figure 5.

The codes for the Flash and HTML challenges both contain a redirect function to the originally requested URL with a hash concatenated to it. If the client is able to correctly execute the challenge code, the proxy will see a separate request with a hash appended to it. If the hash is correct, the new request is allowed to pass through and the hashmap of Greasyspoon is updated to reflect that a particular IP and software agent combination has passed the challenge. The size of the request scripts each average around 280 lines of code.

Since the hash is sent back in plain text it is conceivable that an attacker could simply parse the response for the hash and initiate the correctly formed request if they have knowledge of our system. We can prevent this attack by encrypting the hash with an AES Javascript implementation [3]. The new Javascript provides the code to decrypt the hash and requires the malware to include functions for AES decryption.

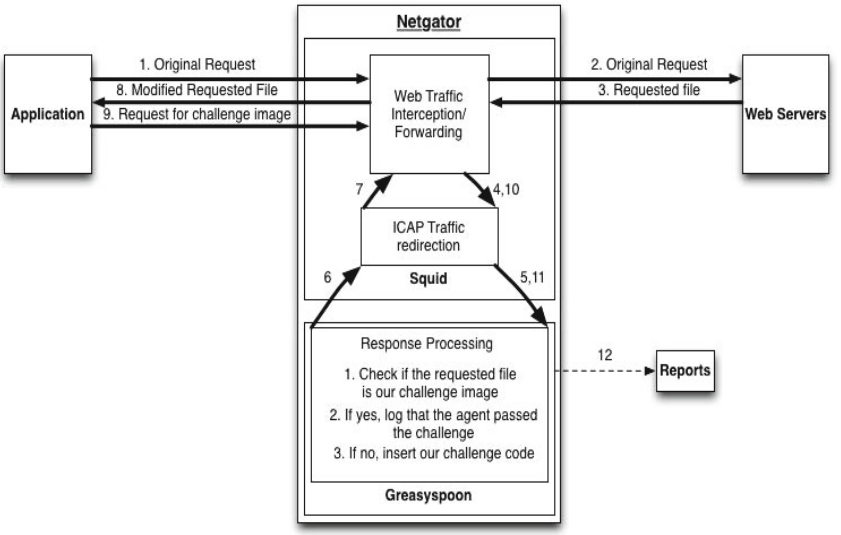



**Fig. 6.** Active Response Challenge Flow

**Challenging Network Responses.** If the data requested is of the type text/ html, the proxy allows the request pass through. When the response comes back for that request, a challenge code is inserted in the response. For instance, an image that resides on the proxy can be embedded in a Javascript challenge code. A Javascript *write* statement tells the browser to include the image via HTML "img" tags. The proxy then looks for requested for this specific image and once it sees one, it knows that the challenge has been completed successfully. Figure 6 shows the control flow for active response challenges.

```
Record number=1
IP=192.168.0.2
OS=Windows XP
App_Name=Firefox
App_Version=3.0.5
UA=Mozilla/5.0 (Windows; U; Windows NT 6.1; fr;
rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
Challenges_Issued=587
Challenge_Responses=587
```

**Fig. 7.** Example Logfile Entry

The processing script has two parts: a request script and a response script. Combined, there is about 300 lines of code. The request processing script first determines if the client is expecting a text/html response or if the request is for our specific challenge image. If the client is expecting a text/html response, an entry of the user-agent string combined with the client IP is written into the hashmap. The original request then goes out to the intended server. If the request is for the challenge image, Greasyspoon searches for a corresponding entry in the hashmap and updates it reflecting that the client has passed a challenge. Once a response for the connection is received, the response processing script probes for an already present entry in the hashmap for the client the response is to be sent to. If an entry is located, it injects the HTML code with the embedded challenge image inside the original response and sends the response back to the client. The entry is revised to show that a challenge has been sent to the client.

The response infrastructure is also responsible for the transformation of the hashmap into a logfile format. An example entry from the logfile is shown in Figure 7. The operating system, application name, and application version are all extracted from the user-agent string. It can help administrators analyze the logs and diagnose a problem should one arise.

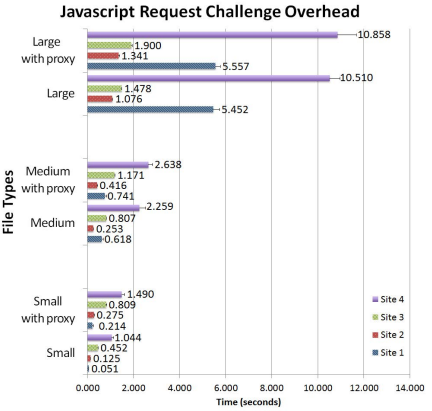
The reason for adapting response challenges is two-fold. First, it allows us to reduce the overhead that might be introduced when enacting the request challenge on each HTTP request. Moreover, blocking every data request would impair our system's scalability and usability. It is conceivable that on a smaller, more confined network the system could be set up to challenge every request, but on a larger infrastructure this would most likely be impractical. Second, it can prevent a malicious agent from downloading an executable that is disguised as an HTML file.

## 5 Experimental Evaluation

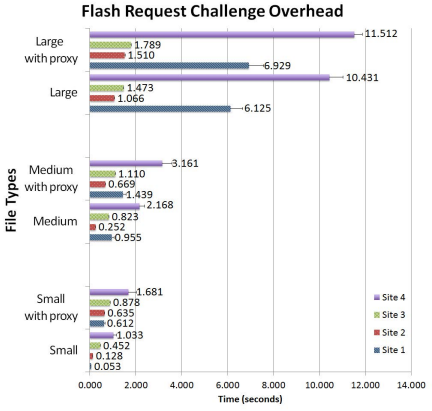
We implemented a prototype NetGator system consisting of a laptop for the client and a Dell server for the proxy. The laptop is a Dell Latitude E6410 with an Intel Core i7 M620 CPU at 2.67 GHz, 8GB of RAM and a Gigabit network interface. The server is a Dell PowerEdge 1950 with two Xeon processors, 16GB of RAM and a Gigabit network interface.

For performance testing, Firefox 3.6.17 is used as the client's browser throughout. To measure how efficiently the server could process the scripts that will be

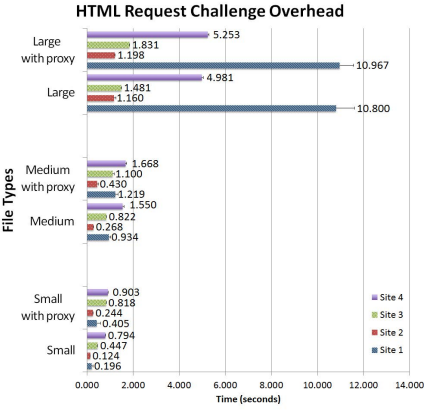
executed on the client's request, a script loops through 10,000 iterations of the request script with the iterations per second being returned. For all the figures in this section, we run this script 30 times and use the average value to determine the capability of our server. The error bars show confidence interval at 95% confidence. For all testing, Squid and Firefox's caching mechanisms are completely disabled.



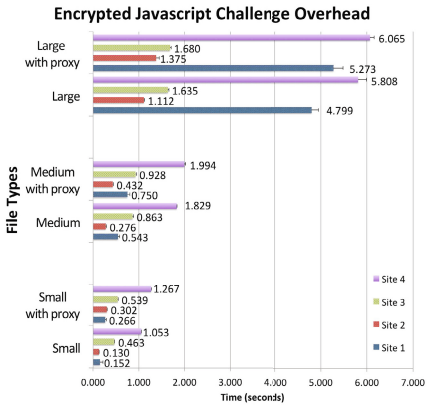

(a) Javascript Challenge End-to-End Latency.




(b) Flash Challenge End-to-End Latency.




(c) HTML Challenge End-to-End Latency.




(d) Encrypted Javascript Challenge End-to-End Latency.

**Fig. 8.** Request Challenge Overhead

## 5.1 Performance Analysis

To evaluate the end-to-end latency of the request challenge, we analyze various types of challenges we create in HTTP download scenarios utilizing PlanetLab [4]

combined with our passive inspection. Different PlanetLab nodes are used from throughout the world using all virtualized hardware. Four nodes (one from the East Coast, one from the West Coast, and two nodes from other continents) are utilized to perform the benchmarking. Executable files of large size (1000KB), medium size (100KB) and small size (10KB) are hosted on each node on an Apache web server. The client then downloads each file thirty times from each of the nodes, both with and without the NetGator proxy. The values of end-to-end latency are determined by the difference in the time-stamp of the packet that starts the initial request before the challenge and the last packet that closes the connection after downloading the file. Figure 8 shows the end-to-end latencies with and without using Javascript, Flash, and HTML request challenges for requesting different sizes of files.

Our experiments show that the end-to-end latency using request challenges is almost negligible to the client. When using Javascript as the challenges, it increases the end-to-end latency by 274 milliseconds in average for all size of file types from four sites. The Flash request challenge has an average latency overhead of 580 milliseconds, while the HTML request challenge introduces 206 milliseconds of latency overhead. Figure 8(d) shows that the encrypted Javascript request challenge has only 174 milliseconds of latency overhead, which is less than the normal Javascript challenge and HTML challenge. We see that enhancing security by encrypting the hash may not increase the end-to-end latency.

To measure the overhead of the response challenges, we perform experiments using a wide-range of websites throughout the country. Baselines are established for each website by performing a simple loading of each of them without the proxy involved. Once these baselines are established, the gateway of the client laptop was changed to be our proxy. Each website is loaded 30 times both with and without the response challenge. In order to establish time, the difference between the time-stamp of the first and last packet in the stream is taken. The results of these experiments are shown in Figure 9. We can see that the response challenge only introduces very small latency overhead.

The request challenge results demonstrate that the overhead remains constant across various file sizes; this means that in terms of percentage the overhead gets progressively smaller as the files downloaded become larger. The response challenge results are even smaller with an average overhead of only 24 milliseconds. With the minimal amount of overhead that our system introduces, it is not perceivable to the user. All of the challenging happens without any interaction from the user, allowing a seamless experience while maintaining the security of a network. Moreover, our proxy is exceedingly efficient in processing the scripts, being able to handle on average approximately 1,200 request scripts per second.

For the malware testing we obtained a set of malware through a custom crafted retrieval mechanism. This mechanism is provided malicious URLs from Malware Domain List and Google. The samples used during testing were solely Windows executables. During our testing to establish what percentage of malware calls out utilizing HTTP/S, we find that none of the malware which used either protocol could overcome our challenge architecture. That sample size equates to

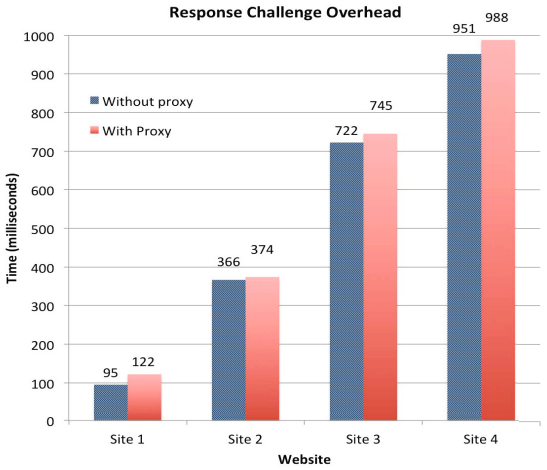



**Fig. 9.** Response Challenge End-to-End Latency

817 malware samples challenged. The typical behavior of the infected systems simply try to re-request the file it has originally sought after only to repeatedly be returned our challenge. We do not observe any false positives in our experiments. The level of false positives will be directly related to how many browsers in a network utilize HTTP/S but do not contain HTML, Javascript, or Flash engines.

## 6 Discussion and Limitations

We assume that malicious agents do not typically access the full software stack of applications that reside on the infected host. We also assume that the malware does not include its own Javascript engine. If malware is forced to implement a full browser agent complete with Javascript/Flash functionality, this would greatly increase the presence of the malware thus increasing its vulnerable to detection. Based on this assumption, the malware will not be able to decode the encrypted hash of the challenge even when the key is passed to the host. Our testing of recent malware samples shows us that the current level of sophistication of malware does not include their own Javascript engines nor the ability to access the full software stack of web browsers present on the system. By issuing each piece of malware our Javascript challenge, we are able to determine that none of the malware tested encompasses the Javascript functionality to overcome our challenge infrastructure.

Our system can challenge either the initial request or the response when considering the trade-offs between security and performance. The request challenge allows NetGator to sever the malware's connection immediately to negate any damage before it happens. This also comes with a cost of slight latency. On the other hand, the response challenge allows the response to return to the requesting agent before it issues the challenge. This dramatically lowers the latency

the user experiences but also allows the original request to complete even if the software agent is detected as malicious later. This method relies on the monitoring of the logs to identify compromised hosts. We could improve its security by utilizing the information about agents that fail the challenge in the response PIC and create a signature to block future outbound connection attempts. Intercepting SSL traffic causes a possible issue in the transparency to users. Our approach to processing HTTPS traffic essentially acts in the same way a man in the middle attack works. Browsers typically identify this behavior and report the suspicious activity to the user. This can be mitigated by the hosts having their organizations certificates installed on the end hosts. Without these certificates the transparency to the users is affected.

If an attacker is aware of the presence of our system, they would likely attempt to craft their communications in a way to evade our detection. An approach that they might take would be to label their communication as a simple non-browser agent. If the agent is not one of the approved applications to transmit across ports 80 or 443, then the connection would be severed. If it is an approved application, it would be challenged in the same way that browsers are. In an enterprise network, there would ideally be a full repertoire of challenges to issue to the various applications that communicate across ports 80 and 443. However, there may be a necessary application for which a challenge can not be crafted. In order for us to create a PIC for a particular application, we require that is has specific functionality that will have an expected response to some request. For agents that do not meet the requirements to create a PIC, a whitelist of servers for these application to communicate with can be constructed and any connections from these applications to other servers would be raised as suspicious. Malware might also utilize a full application (a legitimate web browser) in their communications to correctly pass our challenges. While possible, this increases the likelyhood of being detected due to not being able to rely on their own covert communications. Encompassing a browser's full capabilities would evade our detection and is a limitation of our approach.

## 7 Related Work

Our work is partly inspired by various automatic protocol analysis systems [20,29], which utilize injection of messages to various applications in order to automatically determine how a particular protocol is organized. Instead of injecting messages to test a protocol, we examine a particular application software to prove its identity.

Recent research by Gu et al. [16] is the most simliar to our approach. They aim to detect botnet communication over IRC through a combination of user interaction and probing for expected responses. There are two main differences with our approach. First, we do not expect a human to be behind the communications, nor rely on one at any time to be able to solve our challenges. Second, their paper focuses on detection of malicious botnets, while ours is concerned with verifying the identity of legitimate end-host applications. Our approach is

beneficial in the sense that the signatures (expected responses) of botnets will continue to grow consistently while Netgator only has to establish signatures (challenges) for legitimate applications which will not likely change their functionality over time.

Our work can be compared to techniques used by OS and application fingerprinting programs such as Nmap [21]. The most popular form of real-time browser challenges is to utilize server-side techniques that read browser configuration files [23,26] (Javascript, ASP, etc.), cookie information [22], or search for platform specific components like Flash blockers or Silverlight [12]. Another approach is to search traffic flows for known, specific identifiers like connections to Firefox update servers [30]. Conversely, techniques like the well-known CAPTCHA puzzles attempt to prove the existence of a human user. However, all those methods are disruptive and not transparent to the user.

Traditionally, botnet detection and mitigation systems like BotSniffer [17] have focused on zombies that contact Internet Relay Chat(IRC) C&C servers or utilize IRC-style communication [9]. Unfortunately, botnets have grown in sophistication to use Peer-to-Peer (P2P) and unstructured communication [10,19]. In addition to the traditional techniques such as blacklisting, both signature and anomaly-based detection, and DNS traffic analysis, BotHunter [15] proposes using infection models to find bots, while BotMiner [14] analyzes aggregated network traffic. Our work is a another complementary study utilizing an active challenge technique to distinguish certain types of bots from benign applications.

Data transmission over HTTP/S is very common and consumes the bulk of un-filtered traffic in most organizations. For analyzing packets that contain a payload, deep packet inspection techniques are favored. Signature or anomaly based detection is applied to these packets [10,13,25,27,8,28]. To foil this mechanism, malware may use the same secure protocols that users employ to protect themselves from malicious agents [24,18]. Our approach only analyzes the data content to determine the protocol being transmitted. Once the protocol type is established, we are are concerned solely with the client behind the communications.

## 8 Conclusions

We study an active and in-line malware detection system, called NetGator. Our goal is to be able to detect outbound malware flows for when malware attempts to establish a network connection to a back-end server. Our approach is two-pronged: the passive classification module analyzes network flows to determine the claimed identity of the end-point software that generated the packets. It relies on existing network program signatures to classify end-point applications and drop the unknown requests. Next, NetGator verifies the legitimacy of the end-point software by generating easy to generate and computed in-line Program Interactive Challenges (PICs) based on the functional capabilities supported by the end-point software.

Although our approach can be potentially circumvented by sophisticated targeted malware, we believe that NetGator significantly increases the complexity

of the attack forcing the adversary to perform additional invasive tasks before it can successfully communicate data. On the other hand, NetGator is fully transparent to the user: our experiments demonstrate that it can examine real-time traffic. In addition, the detection results demonstrate the effectiveness of PICs in identifying malware that attempts to imitate the network connection of popular browsers. NetGator introduces an average of 353 milliseconds end-to-end latency overhead using network request challenges and 24 milliseconds using network response challenges.

## References


1. Anti-Phishing Working Group, ADWG 2011 Trends Report, http://apwg.org/reports/apwg_trends_report_h1_2011.pdf
2. Greasyspoon, http://greasyspoon.sourceforge.net/
3. Javascript encryption, http://javascript.about.com/library/blencrypt.html
4. Planetlab, http://planet-lab.org/
5. Squid, http://www.squid-cache.org/
6. Tcpflow, http://afflib.org/software/tcpflow
7. Wireshark, http://www.wireshark.org/
8. Androulidakis, G., Chatzigiannakis, V., Papavassiliou, S.: Network anomaly detection and classification via opportunistic sampling. IEEE Network 23(1), 6–12 (2009)
9. AsSadhan, B., Moura, J., Lapsley, D., Jones, C., Strayer, W.: Detecting Botnets Using Command and Control Traffic. In: Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications, pp. 156–162. IEEE Computer Society (2009)
10. Bailey, M., Cooke, E., Jahanian, F., Xu, Y., Karir, M.: A survey of botnet technology and defenses. In: Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security, pp. 299–304. IEEE Computer Society (2009)
11. Cyveillance. Malware detection rates for leading av solutions (2010), http://www.cyveillance.com/web/docs/WP_MalwareDetectionRates.pdf
12. Eckersley, P.: How Unique Is Your Web Browser? In: Atallah, M.J., Hopper, N.J. (eds.) PETS 2010. LNCS, vol. 6205, pp. 1–18. Springer, Heidelberg (2010)
13. Garcia-Teodoro, P., Diaz-Verdejo, J., Macia-Fernandez, G., Vazquez, E.: Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security 28(1-2), 18–28 (2009)
14. 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 Conference on Security Symposium, pp. 139–154. USENIX Association (2008)
15. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: Bothunter: detecting malware infection through ids-driven dialog correlation. In: SS 2007: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, pp. 1–16. USENIX Association, Berkeley (2007)
16. Gu, G., Yegneswaran, V., Porras, P., Stoll, J., Lee, W.: Active botnet probing to identify obscure command and control channels. In: Computer Security Applications Conference, ACSAC (2009)

17. Gu, G., Zhang, J., Lee, W.: BotSniffer: Detecting botnet command and control channels in network traffic. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium, NDSS 2008. Citeseer (2008)
18. Inoue, D., Yoshioka, K., Eto, M., Hoshizawa, Y., Nakao, K.: Malware Behavior Analysis in Isolated Miniature Network for Revealing Malware's Network Activity. In: IEEE International Conference on Communications, ICC 2008, pp. 1715–1721. IEEE (2008)
19. Karasaridis, A., Rexroad, B., Hoeflin, D.: Wide-scale botnet detection and characterization. In: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets, p. 7. USENIX Association (2007)
20. 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)
21. Lyon, G.: Nmap security scanner (2010)
22. McKinley, K.: Cleaning Up After Cookies (2008)
23. Microsoft Developer Network: How to: Detect browser types and browser capabilities in asp.net web pages (2010)
24. Perdisci, R., Lee, W., Feamster, N.: Behavioral clustering of http-based malware and signature generation using malicious network traces. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, p. 26. USENIX Association (2010)
25. Polychronakis, M., Anagnostakis, K., Markatos, E.: Network-level polymorphic shellcode detection using emulation. Journal in Computer Virology 2, 257–274 (2007) 10.1007/s11416-006-0031-z
26. Schools, W.: Javascript browser detection (2010), `http://www.w3schools.com/js/js_browser.asp`
27. Shon, T., Moon, J.: A hybrid machine learning approach to network anomaly detection. Information Sciences 177(18), 3799–3821 (2007)
28. Thorat, S., Khandelwal, A., Bruhadeshwar, B., Kishore, K.: Payload content based network anomaly detection. In: First International Conference on the Applications of Digital Information and Web Technologies, ICADIWT 2008, pp. 127–132 (August 2008)
29. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: 15th Symposium on Network and Distributed System Security, NDSS (2008)
30. Yen, T.-F., Huang, X., Monrose, F., Reiter, M.K.: Browser Fingerprinting from Coarse Traffic Summaries: Techniques and Implications. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 157–175. Springer, Heidelberg (2009)

# SMARTPROXY: Secure Smartphone-Assisted Login on Compromised Machines

Johannes Hoffmann, Sebastian Uellenbeck, and Thorsten Holz

Horst Görtz Institute (HGI), Ruhr-University Bochum, Germany
firstname.secondname@rub.de

**Abstract.** In modern attacks, the attacker's goal often entails illegal gathering of user credentials such as passwords or browser cookies from a compromised web browser. An attacker first compromises the computer via some kind of attack, and then uses the control over the system to steal interesting data that she can utilize for other kinds of attacks (*e. g.*, impersonation attacks). Protecting user credentials from such attacks is a challenging task, especially if we assume to not have trustworthy computer systems. While users may be inclined to trust their personal computers and smartphones, they might not invest the same confidence in the external machines of others, although they sometimes have no choice but to rely on them, *e. g.*, in their co-workers' offices.

To relieve the user from the trust he or she has to grant to these computers, we propose a privacy proxy called SMARTPROXY, running on a smartphone. The point of this proxy is that it can be accessed from untrusted or even compromised machines via a WiFi or a USB connection, so as to enable secure logins, while at the same time preventing the attacker (who is controlling the machine) from seeing crucial data like user credentials or browser cookies. SMARTPROXY is capable of handling both HTTP and HTTPS connections and uses either the smartphone's Internet connection, or the fast connection provided by the computer it is linked to. Our solution combines the security benefits of a trusted smartphone with the comfort of using a regular, yet untrusted, computer, *i. e.*, this functionality is especially appealing to those who value the use of a full-sized screen and keyboard.

**Keywords:** Web Security, Browser Security, Privacy Proxy, Smartphone, SSL.

## 1 Introduction

Regardless of the passing years witnessing numerous efforts to secure software systems, we are still observing security incidents happening on a daily basis. Attackers have managed to compromise the integrity of computer systems, as techniques such as control-flow attacks on software systems or exploiting logical flaws of applications were developed [2,11,23,25]. Due to the growing complexity of today's systems, it is highly unlikely that we will have trustworthy computer systems without *any* security flaw at our disposal in the near future. We thus need to investigate how certain security properties can be guaranteed despite compromised computer systems being in operation. In this paper, we focus on the threat of stealing user credentials from web browsers. More precisely, the attacker's goal often entails illegal gathering of user credentials

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 184–203, 2013.

such as passwords or browser cookies from an already compromised machine. Several reports on malicious software specialized in stealing credentials provides evidence for the prevalence of this threat (*e. g.*, bots and keyloggers) [9,15].

In practice, we do not only need to worry about the integrity of our personal (home or office) computer, but of each and every machine we interact with and enter our credentials into. Surfing on public Internet terminals, using computers in Internet cafes, or simply browsing the web on a arbitrary in-office or friend's computer can all bring about a number of potential threats. A golden rule of thumb for our safety is that we cannot trust any such machines, or going one step further: we have to assume that they have been compromised. Still, all things considered, protecting the credentials from attackers in such situations proves to be a major challenge: if an attacker has compromised a system, she has complete and full control over the system and can modify the settings in an arbitrary way. Furthermore, she can read all memory contents, the CPU state, and all other information pertaining to the current state of the system in question. Based on the so-obtained data, an attacker can easily reconstruct (user) credentials [12,14] as it becomes harder (if not impossible) to protect the integrity of user personal, login, and other sorts of data.

One idea for improving the integrity of credentials is to use external trusted devices and generate one-time keys (*e. g.*, RSA SecurID [8] or Yubico YubiKeys [27]). Here, even if an attacker has compromised the system, she only obtains a one-time password that cannot be reused, hence the potential harm is greatly limited. Nonetheless, this approach does not support legacy applications and requires changes on the server's side since the one-time password needs to be verified by the server. Furthermore, the compromise of RSA and Lockheed Martin in 2011 [5] has illustrated that such systems can also be periled, provided that the seed files for the authentication tokens can be accessed by a skilled attacker. A different idea is to use an external device with more computational power as a trusted intermediary supporting the authentication routine [4,17,26]. Such approaches typically either require changes on the server side or an additional server that needs to be trusted. As a result, they are typically not compatible with existing systems or trust needs to be put into another device not under the user's control.

In this paper, we introduce a system called SMARTPROXY that protects user credentials when interacting with a compromised machine. The basic idea is to use a smartphone as a special kind of *privacy proxy* as we shall tunnel all network connections of the untrusted machine over the phone and filter all sensitive data there. To be more precise, we intend the user to only enter fake information on the untrusted machine, as we will have our tool substitute this data with the real credentials on the outgoing network connections on the fly. For incoming network connections, SMARTPROXY will also replace different kinds of sensitive data (*i. e.*, cookies) with imitative information, so that a potential attacker operating from the untrusted machine cannot achieve her goals. As a result, a (semi-)trusted smartphone takes care of handling sensitive data related to the user and therefore, the confidential information never reaches the untrusted machine on which an attacker could potentially track and observe it. This is achieved through selective usage of encrypted communication channels via SSL: SMARTPROXY intercepts the phone's outgoing SSL connections and performs a Man-in-the-Middle (MitM) attack, eventually being capable of substituting fake credentials with valid ones. All of

the tool-initiated network connections to the destination server additionally use SSL, meaning that an attacker cannot intercept or eavesdrop on those channels.

Since the phone is a potential target for the attacks, we also make sure that we split the transferred information in such a way that even if an attacker compromises both the machine *and* the smartphone, she at least cannot steal all data at once. This is achieved by encrypting all private data with a user-selected key for each credential. Conclusively, our tool enables a sound and robust method of credentials' handling. We have implemented a fully working prototype of SMARTPROXY for Android-based devices. Our evaluation demonstrates that the overhead introduced by the tool is reasonable: micro-benchmarks indicate that the SSL handshake typically takes less than 42 ms on a current smartphone, while in macro-benchmarks we found a typical overhead of less than 50 percent to the load time for popular websites.

In summary, we make the following contributions:

- We introduce the concept of a smartphone-based privacy proxy that can be used to preserve credentials in spite of the presence of a compromised machine that an attacker controls.
- We have implemented a fully-working prototype and our performance benchmarks indicate that the overhead of SMARTPROXY on typical websites is moderate.

## 2 System Overview

In this section, we describe the attacker model which has been employed throughout the rest of this work. Furthermore, we give a brief overview of our approach to securing user credentials in face of a compromised machine and sketch the high-level idea.

### 2.1 Attacker Model

In the following, we assume that an attacker has compromised a computer system and thus has complete control over it. This allows her to obtain the system state's full information and enables arbitrary modifications. On a practical level, this might be the case when an attacker managed to infect a computer with some kind of malicious software that can steal credentials and/or alter the settings of a web browser. Furthermore, we assume that the attacker cannot break a public key crypto system and symmetric ciphers with reasonable parameters, *i. e.*, we assume that unless the used key is known and out in the open, the used ciphersuites cannot be broken by the attacker. As a special action, the attacker might disable our proxy for the web browser. Doing this will only result in a denial of service attack against the user, as long as the user always only enters fake credentials into the web browser.

A user who wants to utilize the web browser and log in to a given website such as Facebook or Wikipedia needs to have valid user credentials. Essentially, he has no simple means to decide whether the machine he wishes to put to work is indeed compromised or not since it is not a machine under his control (*e. g.*, a public Internet terminal or an arbitrary in-office computer). He may also simply not know how to verify the integrity of the machine in question. Overall, it is a complex problem in itself and we assume that the user cannot efficiently assess whether the endangerment has taken place

or not. Luckily, the user has as an auxiliary means of resorting to his smartphone which he can, to a certain extent, trust. We suppose for now that the smartphone has not been compromised, which implies that the user can run software on the phone without having to worry about an attacker. At the same time, we understand that the user wants to surf the Web on a computer rather than a smartphone since it provides a larger screen and a natural-sized keyboard. In addition, the computer has more computational power, as it is for example capable to render movies quickly and with ease.

Based on these assumptions, the goal of our system can be clearly indicated as enabling the user to log in to a website without vexing about his credentials. We focus on the HTTP and HTTPS protocol, connections with arbitrary protocols over sockets are not considered for now. With our tool in place, the attacker will neither acquire the login credentials (*i. e.*, username, passwords, or browser cookies) nor obtain any information about them. Note that the system cannot protect the web content on the computer as the attacker is already assumed to be able to read all information on that device. For that reason, the attacker can still read as well as modify all content accessed by the user, yet she cannot log in on the website at a later point in time as all valid user credentials are no longer there to be taken away.

In addition, we need to make sure that an attacker cannot obtain credentials by compromising the smartphone. This would imply that our system has a single point of failure. Even if the attacker can compromise both the computer *and* the smartphone at the same time, she remains unable to acquire all valid credentials. By splitting the actual credentials in a clever way (see Section 3.6), we warrant that an attacker can only observe the data which are evidently made use of in real time.

### 2.2 System Overview

As stated above, in order to protect user credentials from an attacker operating a compromised machine, we propose to use a smartphone as a kind of privacy proxy. As a preparatory step, the user needs to configure the web browser on the computer in such a way that it engages the smartphone as a web proxy for HTTP and HTTPS traffic. As we discuss in Section 3.2, there are quite a few different ways for a smartphone being connected to a computer, which leads us to pinpointing an easy way to set up our system. Moreover, the user needs to import a X509 V1 root certificate into the web browser, as we need to intercept SSL connections (as we will explain later on).

Once the machine and the smartphone are connected, the smartphone transparently substitutes fake credentials entered by the user on the compromised computer with valid credentials, which are then sent to the target destination the user wants to visit. The smartphone can perform this substitution by carrying out a MitM attack on the HTTPS connection: our tool intercepts the initial SSL connection attempt from the computer, establishes its own SSL session to the target web server on behalf of the computer, and transparently substitutes the fake credentials with valid ones. These steps require us to launch a fake SSL connection to the web browser on the computer, for which we spoof a HTTPS session pretending to be the targeted web server. We generate an SSL certificate that corresponds to the targeted server using the root certificate imported in the preparation phase. Thus, this certificate will be accepted by the browser and the user

can regard this connection trustworthy. We discuss the workflow of our approach and the MitM attack in more detail in Sections 3.3 and 3.4.

Moving on, the smartphone supplants sensitive information sent from the website to the browser (such as browser cookies) in a way that an attacker cannot obtain them. Our tool successfully alters information on one hand, yet grants the possibility of a web browser usage on the other. Meanwhile an attacker does not have the power to obtain any useful information from the processes taking place. More details about this substitution procedure and its security aspects are available in Section 3.5.

Effectively, we move the handling of sensitive data from the (potentially compromised) computer to the smartphone, which prevents an attacker from stealing sensitive data. De facto, this carries the problem over to another device which we need to protect: the smartphone turns into an attractive target for the attackers, as it now serves as a primary storage space for all the sensitive data. At the end, we need to be fully confident that the information is kept and retained in a way that fully prevents an attacker from accessing it, even if she manages to compromise *both* the computer and the smartphone at the same time. This can be achieved by storing and provisioning information in an encrypted manner. Decrypting this data may only occur strictly on demand, as we explain in detail in Section 3.6. The potential damage is here-limited to the actually-used credentials exclusively.

## 3 Implementation

We now describe the implementation of the ideas sketched above in a tool called SMARTPROXY. More specifically, we explain how the browser is able to communicate with the proxy running on a smartphone and what user interactions are required. We also clarify how the whole process of enabling secure logins on compromised machines is handled by the proxy. We conclude this section with a description of secure storage of private user data within SMARTPROXY, as it is employed to minimize the attack surface.

### 3.1 Software Overview

SMARTPROXY is implemented as an Android application in Java and we use the *Bouncy Castle* library to forge certificates. All services related to SSL are provided by Android's own SSL provider *AndroidOpenSSL*. Our implementation of the HTTP protocol supports both HTTP/1.0 and HTTP/1.1 (we implemented a required subset of the protocol, not the whole specification), multiple concurrent connections, SSL/TLS (for the rest of the paper we simply use the shortened term SSL), and HTTP pipelining of browser requests. The proxy software includes a graphical user interface on the smartphone, mainly utilized to start or stop the proxy and manage security-related aspects of the tool. More precisely, a user can manage trusted and forged certificates, stored cookies, and user credentials within the interface. The proxy itself listens on two different TCP ports, one for the plain HTTP traffic, and a second one for the encrypted HTTPS traffic.

One goal of the design was the minimization of required user interaction since SMARTPROXY should not become yet another complicated task to deal with at the user's end. Nevertheless, some user interaction is unavoidable for normal operation.

For the proxy to be used, it has to be connected to the computer in one of the ways enumerated in the next section. The browser on the untrusted computer then needs to be configured to employ the smartphone as proxy. This simply requires setting the smartphone's IP address and the ports as the proxy address of the browser. If it is the first time for engaging the proxy, an automatically generated X509 V1 root certificate has to be exported from the smartphone (*e. g.*, either through the SD card or by mounting the smartphone as external storage device). It shall later be imported into the web browser as a trusted root certificate destined and valid for signing other websites' certificates.

Once the setup process has been completed, any type of user interaction is required in only two additional cases. First, in the instance where our SSL trust manager is unable to verify a server certificate. In practice, this might happen in several different circumstances, *e. g.*, if the certificate is self-signed, not signed by a trusted authority, or if the certificate is for some reason invalid (*e. g.*, it has expired). Beware that an invalid certificate might equally indicate an attack as we discuss in Section 3.4. If a certificate cannot be verified, the user has to manually examine and perhaps approve the certificate in a dialog on the smartphone. This is a replication of a security model behavior as it can be found in web browsers. If a user accepts the certificate, an exception for this certificate is generated and further on it will be perceived and processed as validated. Second case of user interaction demand is of course linked to the setup of the fake credentials for any website that the user wishes to log in to securely.

In addition to the aforementioned interactions, the user is able to list and delete all stored cookies and edit credentials with their original and substituted values. SMARTPROXY also enables a user to list and delete all forged and manually trusted certifications. For example, it might at one point be necessary to issue a removal of a user-trusted certificate which has been later on proven invalid and shall therefore no longer be used to establish encrypted connections to the corresponding server.

### 3.2 Communication Modes

The communication between SMARTPROXY running on the smartphone and the web browser running on the untrusted computer is possible in several different ways:

**Computer Acting as WiFi Access Point:** A computer provides a wireless hotspot or an ad-hoc wireless network to which the smartphone connects to. The smartphone's IP is displayed in the user interface and it is this address that is used as the proxy address by the web browser running on the untrusted machine. With this setup, all network traffic is "routed back" to the computer and this particular network connection is used for tunneling, as it is likely faster than the network connection available via GSM or 3G networks. Practically, solely rooted Android devices are able to connect to ad-hoc networks because vanilla Android devices do not support this connection mode.

**Smartphone Acting as WiFi Access Point:** The Android OS is capable of serving as an access point for WiFi-enabled computers. For example, a notebook could connect to the smartphone's access point and utilize the proxy running on the phone. In this configuration, all traffic is routed over the smartphone's own Internet connection (*e. g.*, GSM, 3G, but not WiFi because it acts as an AP), and the linked-in computer will not be able to observe *any* altered outgoing Internet traffic.

**USB Cable with USB Tethering:** In this setup, the smartphone is connected to the PC via the USB port. The phone needs to be configured for the USB tethering and the USB network device on the PC will in this case get an IP address from the smartphone. The smartphone's IP address can again be displayed within the user interface and is also used as the proxy address in the web browser running on the computer. This configuration routes all traffic through the smartphone's own Internet connection and the computer does not observe any additional network traffic (similar to the previous case, but this time the WiFi connection may be used). To the best of our knowledge, iOS does not support USB tethering on its own and Windows requires the smartphone's drivers to be properly installed. At the same time, however, most Linux distributions offer this functionality out-of-the-box.

**Smartphone and Computer on Same Network:** If both the computer and the smartphone use the same (wireless) network, then the smartphone is accessible from the computer and the computer can easily use the smartphone as a proxy. Furthermore, the smartphone may use the Internet connection offered by the WiFi network, instead of it's own GSM or 3G connection. If this setup is possible, it is the easiest to use in practice.

Upon familiarizing oneself with the list of communication methods above, one may notice that all these setups do not require any special access rights on either side. Although the first three setups require special privileges on the computer, they are likely to be enabled for all users because setting up a network interface is a common use case for most consumer targeted operating systems. This was an implementation aim for the software to be kept usable in most environments.

### 3.3 Proxy Workflow

We now present SMARTPROXY's workflow and the different steps that are necessary to enable the filtering of sensitive data. During the preparation stage, we need to connect the smartphone to the computer by using one of the methods discussed in Section 3.2. Next, it is necessary to set up the browser on the computer (*i. e.*, configure a proxy within the browser) and start the proxy software on the phone. If this is the first time SMARTPROXY is used, the tool's X509 V1 root certificate must be imported into the web browser. After performing these actions, the actual workflow can start (see Figure 1), which we discuss in the following.

1. SMARTPROXY listens for new network connections.
2. When the user opens a website in the browser on the computer, SMARTPROXY accepts this connection and spawns a new thread which parses the HTTP `CONNECT` statement.
3. SMARTPROXY opens a TCP connection to the desired web server, initiates an SSL handshake (called *target SSL handshake*) and verifies the server's X509 certificate. If it is not trusted or invalid (*e. g.*, the certificate is expired or the attacker on the compromised computer attempts to intercept the connection), the tool ceases the action and notifies the user that an exceptional rule for this certificate has to be generated in the hopes of proceeding with the connection to the selected server.

4. If this is the first time a connection is established for this particular destination, we need to forge the supplied server certificate with our own RSA keypair and store it for later use. The forged certificate is an X509 V3 certificate signed by the proxy's CA certificate. Note that the user has imported this certificate in the browser in the preparatory phase, thus the browser regards this forged certificate as valid.
5. SMARTPROXY responds to the web browser with a plain text `200 OK` HTTP status code and upgrades the unencrypted TCP connection from the browser to an encrypted SSL connection with the forged certificate from the last step (called *local SSL handshake*). The web browser now assumes that it is talking securely with the designated web server. From this point forward, all network traffic between the web browser and the destination web server is encrypted and will be eavesdropped by the proxy. This and the following step are similar to a normal MitM attack except for the signing part. A more in-depth analysis of the MitM attack we have performed is available in Section 3.4.
6. After the connection setup is done, all requests from the web browser are served and filtered between the two endpoints. The filtering performed by our tool substitutes fake credential user data entered on the (potentially compromised) web browser with genuine credentials. Furthermore, we remove cookies and other types of sensitive data whilst hiding them from the web browser, as explained in Section 3.5. In order to provide some feedback to the user as to when user credentials are replaced, the smartphone vibrates and generates a visual output on each substitution.

This workflow corresponds to the SSL-secured HTTP connections, but a very similar approach can be used for plain HTTP connections with only minor modifications. Steps 3–5 are not needed, and solely standard HTTP requests (*e. g.*, `GET` or `POST` operations) have to be processed. We discuss the security implications of plain HTTP requests in Section 5.

### 3.4 Man-in-the-Middle Attack

Our approach relies on a MitM attack for the SSL-secured connections. Under the assumption that the user's computer is compromised, we need to presume that an attacker has total control over the machine and therefore can read and manipulate all data, including the input received from the proxy. This indicates that the SSL connection between the web browser and the proxy after the initial `CONNECT` statement may also be intercepted and is therefore somehow useless. To keep the confidential data hidden from the compromised machine, no encryption between the web browser and the proxy is required, as sensitive data is only transferred in another SSL-secured connection between the proxy and the designated web server. This connection can be guaranteed to remain unreadable by the attacker, even if the traffic is back-routed over the compromised computer. This is ascertained by the verification of the web server's SSL certificate within the SSL connection between the proxy on the smartphone and the target web server.

All these circumstances make it feasible to use no encryption on the connection between the web browser and the proxy, which would in turn decrease the security-pressure placed on the smartphone. However, this is unfortunately not supported by the web browsers we have tested. They instead expect an SSL handshake after each `CONNECT`

statement and an SSL "downgrade" to the NULL cipher is not supported. All things considered, this behavior is clearly highly beneficial for general security reasons.

In order to successfully carry out our MitM attack, we need two RSA keypairs. These keys have a modulus of 1024 bit which is believed to be safe enough, if we reckon the security point of view as the one elaborated on above. We did not choose stronger keys because of our desire to require as little as possible computational power on the smartphone. One of these RSA keypairs is used in an X509 V1 certificate, which is the root certificate of our own certificate authority. The other keypair is used within all forged X509 V3 certificates as the public key. This way, we only need to create two different RSA keypairs and helpfully save computational time when we have to forge a new certificate. The web browsers do not check whether all certificates they see use the *same* public key, thus they are prone to accept them as long as they have a valid digital signature (from our V1 certificate): the browsers verify the complete certificate chain, but use the same keypair for all relevant SSL operations. Interestingly enough, we actually have not intended doing this check-up in our implementation, but discovered it during the testing phase.

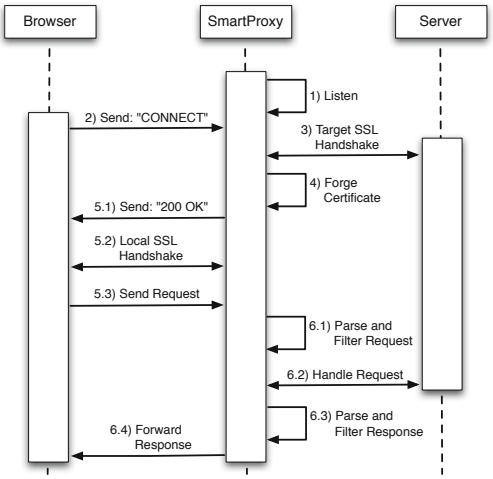



**Fig. 1.** Protocol sequence diagram

In order to keep the number of forged certificates low, hence enabling faster SSL Session resuming (see Section 4.1), step 5 from Section 3.3 generates forged certificates valid for more hosts than the original ones. If an original certificate is only legitimate for the host `a.b.c`, its new counterpart will be valid for the following hosts: `*.a.b.c`, `a.b.c`, `*.b.c`, and `b.c`. These *alternate subject names* are added to the forged certificate for each host which is found in the original one.

If the proxy is accessed over an insecure WiFi connection, the aforementioned encryption between the proxy and the web browser is suddenly a sound cause for concern. Another skilled attacker might sniff this connection and attempt to break the SSL encryption. This would allow her to observe the connection between the web browser and the proxy over the airlink. Such an attacker might not be detected by either side, thus the attacker in question might be able to capture data. However, this secondary attacker does not obtain more data compared to the attacker who has compromised the machine, since both can only observe obfuscated data *after* the substitution by SMARTPROXY took place. An attack on the air link can be easily prevented by increasing the keysize to a strength of 2048 bits or more. It is also possible to generate and use different keys for each forged certificate.

### 3.5 Filtering

In order to transfer all security related information while surfing the web, the proxy has to change some elements in the data stream between the web browser and the web server. This is accomplished by implementing an extendable set of filters which can be attached to each data stream. These filters can in turn be used to change arbitrary HTTP header fields as well as payload data. When wishing to perform a login on a specific website without inserting the real credentials into the untrusted computer, the first action for a user to carry out is to set these accounts up within SMARTPROXY. The account data contains: real username and password, the domain where the login should be fulfilled, and fake username and password. All the above can be inserted into and modified through a GUI. The information is encrypted before it is written into a database (see Section 3.6 for details), and can then be used by our filters. Different filters are applied to the process of data modification, which we describe in the following.

**Password Filter.** Real credentials should not be entered into the user's computer, therefore one has to exercise fake data input sent from the web browser to the server. Certainly, these fake credentials have to be substituted with genuine ones expected by the web server for the login's fulfillment. The Password Filter's function is to find and replace these fake credentials in the data stream. To do just so, the filter searches the `POST` data for the false credentials, determines if the credentials are indeed entered into valid form fields, and then performs the substitution. Furthermore, credentials are only substituted if the domain in the request matches the stored domain in the database.

In order to defeat some attacks, two additional checks are performed before the real credentials are inserted. First, we check if the password was already requested a short amount of time before (*e. g.*, 15 minutes). After a valid login, the user should be identified by some session ID and no credentials should be used anymore, thus this case might indicate an attack. Second, we check if the request contains at least three variables from which two are identical and a fake password. This could mean that someone is attempting to change the password (which might be an attacker, see Section 5). In both cases, SMARTPROXY asks the user for confirmation.

**Authentication Filter.** To support HTTP authentication, we added a second filter that searches the `Authorization` field inside the request's header. In this case, we used the same framework as for the Password Filter, seeing as the user is also required to set up the corresponding fake credentials. Currently, our prototype supports the widespread *Basic Authentication* exclusively and not the lesser used but more secure *Digest Authentication*. To effectuate this functionality, we only had to substitute fake credentials with their genuine counterparts, just as we perform this in the Password Filter (except for them being Base64-encoded in this particular instance). The filter first decodes the data, then checks for a positive match, substitutes the fake credentials, and finally encodes them back before performing the act of sending them to the server. Note that this filter cannot make use of the rate limit for substitutions as the Password Filter since each request will contain the fake credentials which have to be substituted.

**Cookie Filter.** A potential attack against our approach could be a session hijacking attack through cross-site scripting. This event involves a session identifier (which authenticates the user in the eyes of the web server) that is stored in a cookie and can be stolen by an adversary, who can then employ this cookie to take over a complete session. To solve such issues, we modify the cookies sent through our proxy. Cookies are set by the web server with the `Set-Cookie` field in the HTTP header and are sent from the web browser to the web server with the `Cookie` field in the HTTP header. The first field is originating from a web server and has to be altered in such a manner that the data reaching the browser cannot be brought into operation for an attempted session's theft.

This is accomplished by all cookie properties (*e. g.*, VALUE, DOMAIN, NAME, PATH) being stored in a database and the value of cookies being changed to an arbitrary string. For the reverse direction (web browser to web server), the cookie values have to be restored from the database. Lastly, cookies can be created on the client machine directly in the browser with the use of techniques such as JavaScript or Adobe Flash. We decided to let these unknown cookies (for our proxy failed to notice a corresponding `Set-Cookie` HTTP header field) pass through in an unaltered stage because we do not think that an adversary can remodel such "self generated" cookies to be harmful for the user, and to be compatible with websites which depend on this behavior. Once we have dropped them, several websites responded and complained with warnings such as "*Your browser does not support cookies!*".

If we substitute all cookies, we might provoke errors with scripts which use cookie values to, *e. g.*, personalize the website. To overcome this, we only substitute cookies which have a value which is at least eight bytes long and has a high entropy since these cookies likely store sensitive data. Shorter cookies or cookies with low entropy are likely not security sensitive since an attacker could brute-force such cookies. Furthermore, all cookies which contain special strings such as *id*, *sid*, or *session* in the name are forced to be substituted. We assume that all security relevant cookies are protected this way and that personalized website settings are still functional. To be able to ensure the proper working of a website, a black- and a whitelist can be set up for each domain to define which cookies shall (not) be protected by SMARTPROXY.

Since some websites use cookie values to form special requests, we implemented a companion filter for the Cookie Filter to preserve the functionality of websites which depend on this behavior. One example is the Google Chat within GMail, which performs the following request: `GET /mail/channel/test?VER=8&at=COOKIE&i...` Here, the substituted value is meaningless and it hinders the website from operating normally. The purpose of the companion filter is to search for substituted cookie values in requests and to insert the correct value within outgoing requests.

Note that all cookies are only replaced within requests or the `COOKIE` header fields if the destination domain matches the domain which is stored in the database for the corresponding cookie. This further hinders an attacker from stealing cookies.

### 3.6 Personal Data Encryption

The smartphone is a kind of a single point of failure when one is investigating the secrecy of the stored user credentials. If it gets lost, stolen, or compromised, all deposited

credentials must be considered known by a third party. To prevent this from occurring, we store all credentials in an encrypted form in the database in such a way that makes them unlikely to be revealed to a potential attacker. Each genuine username $u_g$ and password $p_g$ is encrypted with a key $k$ derived from a fake password $p_f$ using a *Password-Based Key Derivation Function* (PBKDF2 [18]). The encryption of $u_g$ and $p_g$ is performed with AES in CBC mode.

To improve the security of the genuine data, each fake password should be different. The fact that a fake password is used to derive the encryption key $k$ for a real username and password leads to the fact that the fake passwords must not be stored in the database. This hinders the filters from Section 3.5 to find and replace these strings, as they may then appear almost anywhere in the header or payload, and their appearance may be just as arbitrary. To enable the matching of arbitrary fake password strings, the user enters the fake password in a special format, *e. g.*, `fp_fakepass1_`. That way, the filters have only to look for strings enclosed by a leading `fp_` and a closing `_`. This is a process that can be put in action easily. The intermediate string is used as $p_f$, and $u_g$ and $p_g$ may be decrypted with the latter operation's result. If by chance multiple strings are enclosed in our chosen markers, the corresponding filter will attempt to decrypt $u_g$ and $p_g$ with all found strings as $p_f$, eventually decrypting the credentials. The correct $p_f$ can easily be determined by the use of a checksum.

## 4 Evaluation

We have implemented a fully-working prototype of SMARTPROXY. In this section, we present benchmark results, analyze the overhead imposed by our proxy software, and discuss some test results of using our proxy on popular websites. All benchmarks are divided into two categories, namely micro- and macro-benchmarks. The former measure "atomic operations' (*e. g.*, an SSL handshakes on the smartphone), while the latter are executed to estimate additional load time caused by our tool when accessing several popular websites. The smartphone we used in the testing phase was an *HTC Desire* with *Cyanogen Mod 7.0.3* (Android 2.3.3, not overclocked, using the default CPU governor). Additionally, the same benchmarks were performed on a vanilla *Samsung Nexus S* with Android 2.3.4. All results were comparable, thus we omit them in the following for conciseness reasons. On the client's side, we used *wget* in version 1.12 to automatically establish new connections, and our test smartphone device was connected to the Internet over a wireless connection to a 54 Mbit AP where a downstream of 2.7 MB/s is reachable. We used the `adb` port forwarding technique to launch and maintain a connection between the web browser and the proxy.

### 4.1 Synthetic Benchmarks

A main feature of the proxy is its ability to alter the payload in the SSL-secured HTTP connections, thus we need to evaluate the performance of the necessary steps. All measurement times are averaged over 100 test runs. In order to be able to measure these times without too much interference from the network latency, web server load, DNS lookups and so on, we have applied the benchmarks to a local server which is only

**Table 1.** Micro benchmark for the target SSL handshake (KS = Keysize in bit, AVG = Average Time, SD = Standard Deviation)

| KS | Ciphersuite | AVG [ms] | SD [ms] |
|---|---|---|---|
| 512 | RSA/AES/256/SHA | 29 | 24 |
| 1024 | RSA/AES/256/SHA | 33 | 17 |
| 2048 | RSA/AES/256/SHA | 37 | 9 |
| 4096 | RSA/AES/256/SHA | 90 | 17 |
| 512 | DHE/AES/256/SHA | 84 | 15 |
| 1024 | DHE/AES/256/SHA | 83 | 17 |
| 2048 | DHE/AES/256/SHA | 90 | 17 |
| 4096 | DHE/AES/256/SHA | 124 | 17 |

**Table 2.** Micro benchmark for the local SSL handshake (KS = Keysize in bit, AVG = Average Time, SD = Standard Deviation)

| KS | Ciphersuite | AVG [ms] | SD [ms] |
|---|---|---|---|
| 512 | RSA/AES/256/SHA | 35 | 16 |
| 1024 | RSA/AES/256/SHA | 42 | 20 |
| 2048 | RSA/AES/256/SHA | 90 | 68 |
| 4096 | RSA/AES/256/SHA | 360 | 326 |
| 512 | DHE/AES/256/SHA | 3,734 | 4,422 |
| 1024 | DHE/AES/256/SHA | 3,344 | 4,096 |
| 2048 | DHE/AES/256/SHA | 3,551 | 4,101 |
| 4096 | DHE/AES/256/SHA | 3,670 | 4,115 |

three network hops away. The server is equipped with a Core i7 CPU, 8GB of RAM and was running Ubuntu 11.04 with Apache 2.2.17 using mod_ssl without any special configurations or optimizations.

SMARTPROXY is required to complete two different SSL handshakes for normal operation, hence we tested several configurations for this prerequisite. SSL Session resumption—for an explanation read below—was explicitly disabled for these benchmarks. At first, we tested the speed rate of the handshake to the target server, and evaluated how this depends on different RSA key sizes and ciphersuites. Then we repeated this test for the local SSL handshake. All measurement times are stated without the added-time necessary to validate a certificate, with the exception of a special check designed to investigate if the certificate has already been validated at some prior time. Certificates are validated when they are seen for the first time and then stored for the later use. Each "new" certificate is first checked against those pre-existing ones to determine whether it is already known and valid, and then, provided it is in the clear, it is accepted. This greatly speeds up the process thanks to the fact that the Online Certificate Status Protocol (OCSP) and certificate checks against Certificate Revocation Lists (CRL) are usually time-consuming, which needs to be done only once for each certificate during the proxy's runtime.

The first SSL-related action that occurs when a browser requests a secure connection to a HTTP server is a client handshake from the proxy to the designated host, necessary to establish a MitM controlled connection between these devices. As per our tests, this takes approximately 33 ms for a 1024 bit strong RSA key with our own server. Table 1 provides an overview of the benchmarking results for establishing a secure channel between the proxy and our local test web server, all for different setups. What we discuss below is that these handshakes were all completed in an acceptable time, even with a 4096 bit strong RSA key, and, equally, for the ephemeral Diffie-Hellman (EDH or DHE) ciphersuites, which are generally slower.

In the second step, the certificate presented by the web server has to be forged in order to establish a secure and trusted connection between the web browser and the proxy. The average time to forge and sign such a certificate is 150 ms, which is sufficiently fast

and needs to be only performed once for each accepted certificate. Operations like persisting a new certificate in a keystore are also here-contained.

During the third and final step, the web browser upgrades its plain text connection to the proxy to an SSL-secured connection-type. This implies an SSL server handshake on the proxy involving a forged certificate. Once again, this is a pretty fast operation which takes only 42 ms on average for a 1024 bit strong RSA key. Table 2 provides an overview of how long the local SSL handshake takes for several configurations. In order to accomplish decent times for the local handshakes, it is crucial to choose the right ciphersuites. While plain RSA handshakes operate fast enough even for a 2048 bit key, a noticeable delay is caused by a 4096 bit key. Conversely, if the web browser chooses EDH ciphersuites for the session key exchange, the time to complete such a handshake might be pinned down as anywhere from 1 to up to 10 or more seconds for each handshake—even with the same certificate, which explains the high standard derivation for these tests. EDH handshakes are more secure, yet evidently and understandably more expensive than plain RSA handshakes. Still, normally they do not reach the factor of 100 which we measured. We have reasons to believe that this operation is not well optimized in the used *OpenSSL* stack on Android, albeit it employs the native code in order to do the expensive operations. The fact that SSL handshakes with EDH ciphersuites to the target server are much faster indicates that this code lacks optimizations for the server mode of the handshake, cf. Section 4.2. We believe that it takes a relatively long time to compute the EDH parameters and this phenomenon would explain the vast amount of time it takes to complete a local handshake with EDH ciphersuites. Conclusively, to enable fast local SSL handshakes, we only accept the following secure ciphersuites for these cases: `RSA/RC4/128/SHA`, `RSA/RC4/128/MD5`, `RSA/AES/128/CBC/SHA`, and `RSA/AES/256/CBC/SHA`. Nevertheless, all available secure (EDH) ciphersuites are enabled for the SSL handshake to the web server.

In order to accelerate the expensive SSL handshakes, SSL and TLS support the mechanisms called *Sessions*. Sessions are identified with a unique ID by the client and the server. Once a completed handshake is bound to such a Session, it will not happen again when the Session is still valid. The client and the server may both reuse a Session, and the already negotiated cryptographic parameters that come with it. This feature can greatly facilitate the establishment of new connections. As an initial handshake from the first step is completed, a new connection to our web server is established within under 30 ms on average from the proxy. A standard estimate for a resumed local server handshake to take place is about 40 ms.

We did not benchmark the SSL operations for different websites separately because all operations, except the SSL handshake to the web server, use the same static key materials in the first step which results in almost the same benchmark for each operation.

### 4.2 Real-World Benchmarks

We have selected the global top 25 websites as they are listed at `www.alexa.com` to be our test sites for additional benchmarking, yet we omitted "duplicate" sites like `google.co.jp` and `google.de`, as well as all the sites we were unable to understand content-wise due to its language (despite the attempted use of website translators), *e. g.,* `qq.com`. All the web pages tested are listed in Table 3. The main feature of the

**Table 3.** Evaluation results for global Top 25 websites from alexa.com without duplicates and sites that do not support SSL (KS = Keysize, HS = Handshake, SD = Standard Deviation, LT = Load Time, OH = Overhead)

| | Micro Benchmarks | | | | Macro Benchmarks | | |
|---|---|---|---|---|---|---|---|
| website | KS [bit] | Ciphersuite[a] | HS [ms] | SD [ms] | LT [s] | OH | Login |
| google.com | 1024 | RSA/RC4/128/SHA | 91 | 49 | 0.95 | 23% | ✓ |
| facebook.com | 1024 | RSA/RC4/128/MD5 | 424 | 103 | 1.37 | 58% | ✓ |
| youtube.com | 1024 | RSA/RC4/128/SHA | 71 | 8 | 2.93 | 20% | ✓ |
| yahoo.com | 1024 | RSA/RC4/128/MD5 | 192 | 52 | 4.34 | 26% | ✓ |
| wikipedia.org | 1024 | RSA/RC4/128/MD5 | 363 | 169 | 5.11 | 50% | ✓ |
| live.com | 2048 | RSA/AES/128/SHA | 595 | 31 | 1.60 | 23% | ✓ |
| twitter.com | 2048 | RSA/RC4/128/MD5 | 400 | 97 | 3.66 | 17% | ✓ |
| linkedin.com | 2048 | RSA/RC4/128/MD5 | 426 | 72 | 1.93 | 78% | ✓ |
| taobao.com | 1024 | RSA/RC4/128/MD5 | 2,716 | 2,817 | 34.82 | 154% | ↯[b] |
| amazon.com | 1024 | RSA/RC4/128/MD5 | 263 | 19 | 2.24 | 18% | ✓ |
| wordpress.com | 1024 | DHE/AES/256/SHA | 527 | 52 | 16.66 | 204% | ✓ |
| yandex.ru | 1024 | DHE/AES/256/SHA | 274 | 50 | 5.75 | 260% | ✓ |
| ebay.com | 2048 | RSA/RC4/128/MD5 | 587 | 51 | 1.49 | 46% | ✓ |
| bing.com | 1024 | RSA/RC4/128/MD5 | 52 | 25 | 0.62 | 142% | ✓ |

[a] DHE with RSA and AES in CBC mode.
[b] We were not able to create a login for this website because we could not translate the content of this site.

proxy is its ability to withhold crucial information such as user credentials and cookies, cf. Section 2. This functionality is tested on all popular websites. Whenever it was possible, we created a login and tested the functionality. We found out that our proxy successfully substituted fake credentials with genuine ones on all tested websites.

We measured SSL handshakes for real-world sites (micro-benchmarks) and again used *wget* for these tests. *Firefox* 5.0 with the *Firebug* 1.8.1 plugin was used to measure website load times (macro-benchmarks). No caching was undertaken between each page reload and we additionally invalidated all SSL Sessions after each connection for the micro-benchmarks. The certificates are validated by mirroring the method described in the previous section. Table 3 lists all times and the implied overhead.

To keep the impact of unknown side effects low, we trialled all websites in a row and repeated this 100 times on a rolling basis. In order to measure the overall overhead (macro-benchmarks), we tested each website five times and calculated the load time, comparing the results for when the proxy was used versus the same action without it.

It is vital to note that these times include a lot of operations with timings which we cannot influence or control. These include the web servers load, the time needed to look up the IP address, certificate checks (OCSP/CRL), as well as general network latency. Overall, the SSL handshake from our proxy to the web server is fast enough not to hinder the user's experience. This is interesting because the SSL Sessions can be resumed, the feature which was explicitly disabled for the micro-benchmarks testing here (see Section 4.1). As this is the time it takes for the initial handshake, it does not

have a huge impact on the page load times for the subsequent handshakes. One notable exception is `taobao.com`, but this page was generally slow on all tested devices.

Table 3 shows that the total overhead is reasonable when SMARTPROXY is used as a privacy proxy. While the tool has a certain overhead, it does not in any case makes it unacceptable to surf the Web. For the majority of the tested sites, the overhead is less than 50%, which means that the web page will load up to 1.5 times slower. These times refer to a complete web page load with an empty cache. When the web page is accessed with the proxy and the web browser uses its cache, the overhead is sometimes not even noticeable by a user. Some sites such as `twitter.com` and `yandex.ru` have a larger infrastructure and several SSL handshakes may be needed in order to load a single page, for the content is served from many different hosts with different SSL certificates. Our approach to add additionally derived alternate subject names to the forged certificates, cf. Section 3.4, may reduce the amount of SSL handshakes which are needed to be performed, although this depends on the infrastructure of the specific website, as the completely different overhead for these two web pages shows in Table 3.

Another aspect which has a certain impact on the overhead introduced by SMARTPROXY is the method used by the web servers to handle the connections. If they close a connection rather than allowing our tool to reuse it, we need to establish a new connection to the proxy and to the web server, which implies *two* additional SSL handshakes. This takes time and is clearly visible in the measurement results for `wordpress.com`, a site that exhibits this behavior. This is obviously an implementation detail, but our prototype currently allows only "pairs of connections": namely one connection established from the web browser is tied to one connection to a designated web server.

Aside for "normal web surfing", we evaluated how fast we are able to download large files. The average download speed with SMARTPROXY enabled is only slightly slower than the download without the proxy. We have seen download rates that differ by only a few KB/s up to some 100 KB/s. This of course depends on the accessed server and the WiFi connection's quality. The fastest download speed achieved with the proxy enabled was 2.7 MB/s. To complete our evaluation, we did some research into a selection of the video portals to check if they work correctly. The video portals typically make use of many different web technologies to stream the video content to the browser. When performing our tests, we have not identified any problems on the three most popular video portals from the global Alexa ranking.

## 5 Limitations

We now discuss several limitations of our solution and at the same time also sketch potential future improvements to strengthen SMARTPROXY. As already explained in Section 3.1, the smartphone may route all its traffic back to the potentially compromised computer, as this is the machine engaged as the Internet access point (default gateway). Clearly, this poses a security problem in case that the web browser uses the proxy for plain HTTP websites. Because all HTTP traffic is unencrypted, an attacker may read and alter all traffic on the compromised computer. This includes all data that have been formerly replaced; or, in other words, the genuine credentials and cookies. The current implementation does not check for this scenario. However, all SSL-secured

connections are not affected because a MitM attack would be detected by the proxy running on the smartphone—as long as the PKI is trustworthy. Furthermore, we could extend SMARTPROXY to automatically "upgrade" each HTTP to a HTTPS connection whenever possible, thus preventing this attack scenario. Another potential weakness is that the login procedure on a website could perform some kind of computation during the login process with techniques such as JavaScript, Java, Adobe Flash, Active-X or other similar technologies. These procedures could compute arbitrary values which are in turn sent to some unknown destinations. Currently, we do not support these setups. Our cookie filtering approach might also cause trouble on some websites due to over- or underfiltering. We have not found any problems during our tests and evaluation but can not guarantee that it always works as expected. Some websites might require the use of the white- and blacklist feature.

As the attacker is able to interact with all the content of a web site, she is able to perform so called *Transaction Generator Attacks* [16]. This means that she can make all actions on behalf of the user, *e. g.*, make orders in web stores, post messages, and so on. This cannot effectively be prevented by SMARTPROXY, as it cannot tell by any means which requests are legit and which are not. Jackson *et al.* propose some countermeasures [16], which could be included in SMARTPROXY. The best defense SMARTPROXY offers for now is that the user can always see what is going on as all requests are visible on the smartphone's screen. The user might simply cut the connection to the proxy if something looks suspicious. In this case, the attacker might disable the usage of the proxy, but will not be able to proceed, as no credentials and cookies are available to her.

We did not carry out a user study to see how successful people are able to set up SMARTPROXY. We are aware that people often have difficulties understanding what is going on with security related problems (see for example [6]). As long as the users remember to never input real credentials into the computer while they use SMARTPROXY, they can at least be assured that their credentials are safe.

Finally, SMARTPROXY is not yet really scalable. Each web browser which shall be used with the proxy has to be set up. This includes importing the root certificate for SSL secured websites. Albeit the certificate is not crucial if the link between the smartphone and the computer is secure—*e. g.*, when connected with the USB cable—, it greatly enhances the user experience as otherwise each SSL secured website will enforce a certificate warning from the browser.

## 6 Related Work

Research in this area to date was focused on the attempts to solve similar problems as we discuss in the following. Mannan and Oorschot [22] proposed an approach to secure banking transactions by using a (trusted) smartphone as an additional trust anchor. In a very similar approach, Bodson *et al.* [7] proposed an authentication scheme where users have to take pictures of QR codes presented on websites in order to log in securely. The picture taken is then sent over the smartphone's airlink to the web server for the authentication to be performed there externally. To fulfill their needs, both approaches had to implement server side changes. Conversely, one of our main goals is to only alter the client's side by adding a self-signed root certificate to a browser and configuring the smartphone as a HTTP/HTTPS proxy.

Hallsteinsen *et al.* [13] also used a mobile phone to obtain a unified authentication. This approach is based on one-time passwords on the one hand and the need for a working GSM system on the other. In this research, an additional Authentication Server is connected to both the GSM network and the service provider which the user wants to authenticate against. The authentication requests are sent via short messages (SMS) in this setup. By doing this, the authors mitigate the risk of MitM attacks, but have to trust the safety of the user's computer, which in our scenario might as well be compromised.

Balfanz and Felten [3] developed an application that moves the computation of the e-mail signatures to a trusted mobile device so that the untrusted computer can be used without revealing the key to the intermediary computer. However, this approach is limited in the application scope.

Perhaps closest to our idea is *Janus*, a Personalized Web Anonymizer [10]. The idea behind Janus is to enable simple, secure and anonymous web browsing. Credentials and email addresses are automatically and securely generated with the help of a proxy server for HTTP connections. The proxy assures that only secure credentials and no identifying usernames and email addresses are sent to the web server. In contrast to our work, Janus does only support SSL secured connections from the proxy to the web server and the proxy resides on the client machine, which does not fit our attacker model. It does also not handle other security relevant information like cookies. Finally, Ross *et al.* [24] proposed a browser plugin which automatically generates secure login passwords for the user. The user has to enter a "fake password" which is prefixed with `@@` and which is replaced by the plugin by some secure password. This approach is similar to our credential substitution process, but does not provide a remedy against compromised machines.

## 7 Conclusion and Future Work

In this paper we introduced an approach to protect user credentials from an eavesdropper by taking advantage of a smartphone that acts as a privacy proxy, *i. e.*, the smartphone transparently substitutes fake information entered on an untrusted machine and replaces it with genuine credentials. We showed that it is possible to enable secure logins despite working on a compromised machine. The steps for achieving the desirable output are relatively easy, as they mainly require connecting the smartphone to the computer and configuring the web browser so that it uses our proxy. Compared to previous work in this area, we do not need a trusted third party that performs the substitution, but everything is handled by the smartphone itself. We have implemented a fully-working prototype of our idea in a tool called SMARTPROXY. The overhead is reasonable and often even unnoticeable to the user as demonstrated with our benchmarking results. Furthermore, we evaluated the security implications of the setup and we are convinced that our solution is beneficial for many users.

In the following, we briefly discuss future work. First, there could be a better connectivity between a smartphone and a computer. A type of an automatic wireless connection would be helpful, and the use could potentially be made of the Bluetooth protocol. We did not look into this aspect because the current connectivity scope was deemed sufficient for the prototype. It would be advantageous to see some kind of "Reverse

Tethering" support on Android, like the ActiveSync feature that Windows Mobile offers. This could enable effortless use of the fast Internet connection of the computer. Furthermore, a guarantee of a proper detection of cases when the plain HTTP traffic is routed back to the computer on which the web browser runs would be essential in order to prohibit vulnerable setups of this sort.

Additional filters could be implemented to hide content from web browsers, for example by some type of blacklist to substitute valuable data with fake information. Examples could include names, addresses, credit card and social security numbers, and even real credentials. This would also avert "mirror attacks" where an attacker attempts to send substituted real data back to the web browser through the proxy to get hold of them. Although the attacker would need access to the web server to enable such mirroring and would already have access to the data, such an attack would be harder for the attacker. As the filter system is pluggable, this can be implemented rather easily but was not done for our prototype at this stage.

Last but not least, the speed of the SSL handshakes could be improved. Although it is already pretty fast for most use cases, this is undoubtedly a desirable direction of improvements in the future. There are some existing solutions which are known to reflect and cater to this need, but they are currently unavailable on the Android OS. One example would be what Google proposes in the instances of *False Start* [21], *Next Protocol Negotiation* [19], *SPDY* [1] and *Snap Start* [20]. These solutions are currently tested by Google in selected applications and servers. If they prove successful, they will be hopefully integrated into Google's Android.

**Acknowledgments.** This work has been supported by the Federal Ministry of Education and Research (grant 01BY1020 – MobWorm), and the DFG (Emmy Noether grant *Long Term Security*).

## References

1. SPDY, `http://www.chromium.org/spdy` (accessed November 04, 2011)
2. One, A.: Smashing the Stack for Fun and Profit. Phrack Magazine 49(14) (1996)
3. Balfanz, D., Felten, E.W.: Hand-Held Computers Can Be Better Smart Cards. In: USENIX Security Symposium (1999)
4. Clarke, D., Gassend, B., Kotwal, T., Burnside, M., van Dijk, M., Devadas, S., Rivest, R.: The Untrusted Computer Problem and Camera-Based Authentication. In: Mattern, F., Naghshineh, M. (eds.) PERVASIVE 2002. LNCS, vol. 2414, pp. 114–124. Springer, Heidelberg (2002)
5. Drew, C., (The New York Times): Stolen Data Is Tracked to Hacking at Lockheed, `http://www.nytimes.com/2011/06/04/technology/04security.html` (accessed November 04, 2011)
6. Chiasson, S., van Oorschot, P.C., Biddle, R.: A usability study and critique of two password managers. In: USENIX Security Symposium (2006)
7. Dodson, B., Sengupta, D., Boneh, D., Lam, M.S.: Secure, Consumer-Friendly Web Authentication and Payments with a Phone. In: Gris, M., Yang, G. (eds.) MobiCASE 2010. LNICST, vol. 76, pp. 17–38. Springer, Heidelberg (2012)
8. EMC Corporation. RSA SecurID, `http://www.rsa.com/node.aspx?id=1156` (accessed November 04, 2011)

9. Franklin, J., Perrig, A., Paxson, V., Savage, S.: An inquiry into the nature and causes of the wealth of internet miscreants. In: ACM Conference on Computer and Communications Security, CCS (2007)
10. Gabber, E., Gibbons, P.B., Matias, Y., Mayer, A.: How to Make Personalized Web Browsing Simple, Secure, and Anonymous. In: Luby, M., Rolim, J.D.P., Serna, M. (eds.) FC 1997. LNCS, vol. 1318, pp. 17–31. Springer, Heidelberg (1997)
11. gera: Advances in Format String Exploitation. Phrack Magazine 59(12) (2002)
12. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold-boot Attacks on Encryption Keys. Commun. ACM 52, 91–98 (2009)
13. Hallsteinsen, S., Jorstad, I., Van Thanh, D.: Using the mobile phone as a security token for unified authentication. In: Proceedings of the Second International Conference on Systems and Networks Communications, ICSNC (2007)
14. Henecka, W., May, A., Meurer, A.: Correcting Errors in RSA Private Keys. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 351–369. Springer, Heidelberg (2010)
15. Holz, T., Engelberth, M., Freiling, F.: Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 1–18. Springer, Heidelberg (2009)
16. Jackson, C., Boneh, D., Mitchell, J.: Transaction generators: root kits for web. In: USENIX Workshop on Hot Topics in Security, HotSec (2007)
17. Jammalamadaka, R.C., van der Horst, T.W., Mehrotra, S., Seamons, K.E., Venkasubramanian, N.: Delegate: A Proxy Based Architecture for Secure Website Access from an Untrusted Machine. In: Annual Computer Security Applications Conference, ACSAC (2006)
18. Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0 (2000), http://tools.ietf.org/html/rfc2898
19. Langley, A.: Transport Layer Security (TLS) Next Protocol Negotiation Extension (2010), https://tools.ietf.org/html/draft-agl-tls-nextprotoneg-00
20. Langley, A.: Transport Layer Security (TLS) Snap Start (2010), http://tools.ietf.org/html/draft-agl-tls-snapstart-00
21. Langley, A., Modadugu, N., Moeller, B.: Transport Layer Security (TLS) False Start (2010), https://tools.ietf.org/html/draft-bmoeller-tls-falsestart-00
22. Mannan, M., van Oorschot, P.C.: Using a Personal Device to Strengthen Password Authentication from an Untrusted Computer. In: Dietrich, S., Dhamija, R. (eds.) FC 2007 and USEC 2007. LNCS, vol. 4886, pp. 88–103. Springer, Heidelberg (2007)
23. Nergal: The Advanced return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine 58(4) (2001)
24. Ross, B., Jackson, C., Miyake, N., Boneh, D., Mitchell, J.C.: Stronger password authentication using browser extensions. In: USENIX Security Symposium (2005)
25. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In: ACM Conference on Computer and Communications Security, CCS (2007)
26. Wu, M., Garfinkel, S., Miller, R.: Secure Web Authentication with Mobile Phones. In: DIMACS Workshop on Usable Privacy and Security Systems (2004)
27. Yubico. YubiKey - The key to the cloud, http://www.yubico.com/products-250 (accessed November 04, 2011)

# BISSAM: Automatic Vulnerability Identification of Office Documents

Thomas Schreck, Stefan Berger, and Jan Göbel

Siemens CERT, D-80200 Munich, Germany

**Abstract.** In recent years attackers have changed their attack vector from the operating system level to the application level. Particularly, attackers concentrate their efforts on finding vulnerabilities in common office applications such as Microsoft Office and Adobe Acrobat. In this paper, we present a novel approach to detect *and* identify the actual vulnerability exploited by a malicious document and extract the exploit code itself. To achieve this, we automatically extract from a security patch information about which code fragments were changed. During the analysis of a document, we open the document using the appropriate application, log the execution path, and automatically identify embedded malicious code using dynamic binary instrumentation. Then both pieces of information are used to determine whether a malicious document exploits a known security flaw and, if so, which vulnerability is targeted.

## 1 Introduction

Over the last years authors of malicious software (malware) have increasingly targeted vulnerabilities in client applications [4], such as document readers, web browsers, or media players in order to compromise machines. Most of these applications use complex data structures, which allow the embedding of code, such as JavaScript in the case of Adobe's Portable Document Format (PDF), and, additionally, provide different kinds of Application Programming Interfaces (APIs) to control the way documents are displayed. These complex data structures and rich functionality make such applications prone to vulnerabilities. Especially office documents have received much attention of today's attackers, since the corresponding applications are widespread and frequently contain vulnerabilities.

### 1.1 Motivation

Recently, several security tools have emerged to analyze the threat of maliciously prepared office documents [1,8,5] and determine whether a document is malicious. However, for corporate network administrators, the prime objective after detecting a malicious office document within the infrastructure is to find out, which vendor patch closes the vulnerability exploited by the document. Up to now, research that focuses on the automatic determination of security

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 204–213, 2013.

patches solely relies on manually created signatures (see Section 2.3). The major drawback of this solution is that manually crafting signatures is a complex, time consuming, and continuous task.

In an enterprise environment, new security patches are not simply rolled out after publication but are typically tested before being deployed, to avoid recovering from patches that break the system. Thus, knowing which patch closes an actively exploited security vulnerability is a valuable information as it helps to prioritize what patches need to be tested first in order to be deployed in a timely manner.

For this reason, we need to be able to distinguish malicious documents from benign ones and determine the vulnerability that is to be exploited in a fast and reliable manner. In detail, we need to be able to extract information whether (1) the exploit targets a known vulnerability or (2) whether it is a so-called *zero-day* exploit, i.e., malicious code that aims at exploiting a security issue not known to the public at this point in time. In the first case, we want to be able to provide information about the required patch for a specific application. In the latter case, we are able to gain knowledge about new security flaws for which a protection has yet to be established. In summary, we want to be able to determine in a single step whether a received malicious document is really a threat to the current infrastructure patch-level.

A similar approach is used by next-generation network intrusion detection systems, which try to determine whether monitored exploit attempts against network services could be successful based on the patch-level of the attacked systems. Such information is needed as it greatly reduces the number of serious incidents an administrator has to deal with every day. Thus, the presented prototype system is not intended to be used by end-users, but as a platform for security administrators to help prioritize patch testing and filtering of incidents concerning malicious documents that target already patched vulnerabilities.

### 1.2 Contribution

In this paper, we present a novel approach to automatically determine whether exploit code in office documents targets a vulnerability for which an update already exists or a new security flaw which requires further analysis. In this context, our approach provides information about the specific vulnerability that is exploited and thus which update is required at a system in order to be protected. We implemented this method in a tool called *Binary Instrumentation System for Secure Analysis of Malicious Documents* (BISSAM), which focuses on Microsoft Office 2003 and 2007 running on Microsoft Windows XP, but this approach can be also used with other applications. We use Intel's dynamic binary instrumentation tool *Pin* [3] to follow the code execution during runtime and dump trace information in case suspicious behavior is detected. The resulting data is then used to determine the matching patch by querying a continuously updated database that contains binary difference information for all patches of a certain application. As a result, we are able to facilitate the process of incident

response regarding malicious office documents in a fast and reliable fashion. Our main contribution is the technique to automatically identify the exploited vulnerability of a maliciously formed office document. We evaluated our approach using several documents, received from real world attacks.

## 2 Methodology Overview and Related Work

In this section, we briefly review the problem of analyzing malicious office documents and determining the exploited vulnerability. Therefore, we begin with an abstract overview of our approach.

### 2.1 Problem Definition

Considering the large number of client applications that can be exploited by malware to take control of a system, it is mandatory to gain knowledge on whether a malicious document poses a real threat to a corporate network or not. Thus, the problem that needs to be solved comprises two steps, detect malicious code embedded in documents and identify the actual vulnerability to be exploited.

Most of today's security tools that are used for malicious document analysis solely rely on signature based heuristics to identify both the malicious code and the vulnerability that is exploited. Moreover, the creation of such signatures is usually performed manually, which is a time consuming, fault-prone, and complicated process. To improve this situation, a new method is required that involves automated and dynamic detection and identification of malicious documents.

The general approach to detect malicious behavior of documents we present in this paper is generic, i.e., it can be used for all kinds of document formats. However, we focus on the detection and identification of malicious office documents here, since a lot of malware exists that targets the according applications.

### 2.2 System Overview

The process of software vulnerability identification implemented by BISSAM, consists of two subsequent parts: *Automatic Exploit Detection* and *Vulnerability Identification*. Apart from these two parts, we also require an automated signature generation and storage process to identify vulnerabilities. This step is performed in the *Signature Generation* system on a regular basis by querying vendor patch information on the Internet. The complete system overview is illustrated in Figure 1.

During *Signature Generation* we extract the vendor's updates and check which files of the particular office application have been modified. Afterwards we extract the binary differences between these files and the corresponding files of a standard office application installation using a binary difference utility. The resulting difference files are further processed and ranked using heuristics to determine whether the change is security related or not. The resulting binary

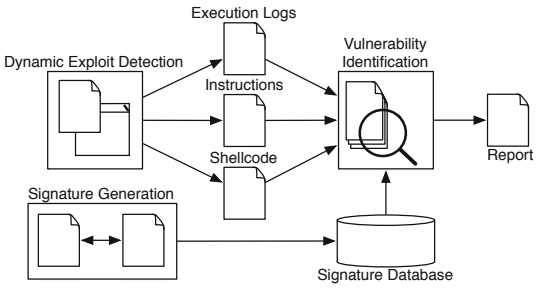



**Fig. 1.** Schematic overview of our approach to analyze malicious documents and extract vulnerability-specific information

difference files are created for each major and minor application version, i.e., we create signatures for Microsoft Office 2003, Office 2003 Service Pack 1, Office 2007 and Office 2007 Service Pack 1. All resulting signatures are stored in a central database to support the vulnerability identification step.

In the first analysis step *Automatic Exploit Detection*, we analyze each malicious office document in a controlled environment. The controlled environment comprises several sandboxes that each contain different versions, patch-levels, and service packs (a bundle of patches) of a particular office application. Then we execute the appropriate office application, such as Microsoft Word, and load the document. Afterwards, we use dynamic binary instrumentation to monitor the program execution. We log the program trace to detect whether malicious code is executed using different heuristics. In case malicious code is detected the application is interrupted and all created log files are copied to the system to identify the exploited vulnerability.

In the second step *Vulnerability Identification*, we use the previously collected log files to determine possible locations in the program execution trace that point to vulnerability exploitations. We compare each basic block with the ones stored in our signature database. A *basic block* (BBL) is a part of code within a program adhering to certain properties. A BBL has one entry point, no code in it is the destination of a jump instruction, and one exit point. We only consider basic blocks for which an update has made security relevant changes. This is determined by heuristics, e.g., if insecure function calls like *strcpy()* are replaced.

The resulting list of basic blocks that match basic blocks found in the database is ranked according to the distance of the basic block from its location in the program execution to the end of the trace and the *match rate* of the signature that matched. Match rates are generated during the signature generation step. This process results in a ranked list of security updates that fit the vulnerability a malicious office document is trying to exploit. In case no update is found we assume the malicious code is targeting an unknown security flaw, a so-called *zero-day*.

### 2.3 Related Work

Next to BISSAM only Officecat [8] and OffVis [5] provide functionality to identify security flaws. However, they use manually created vulnerability signatures, thus the detection quality depends on the signatures. Further, new signatures have to be distributed each time a new vulnerability is detected. Our approach creates new signatures automatically based on the vendor's updates. At the time of this writing, Officecat and OffVis only provide signatures for vulnerabilities found between 2006 and 2009.

## 3 Approach

In order to correctly identify which update is responsible for correcting a certain vulnerability, we created a database containing signatures to link updates to code blocks as extracted by Pin during the automatic exploit detection process. Signatures are created with the help of the tool *DarunGrim* [6]. DarunGrim supports the process of detecting differences in binary files, also known as *binary diffing*. We define a *Signature* as a changed, removed, or added BBL by a security update. One update may result in various signatures.

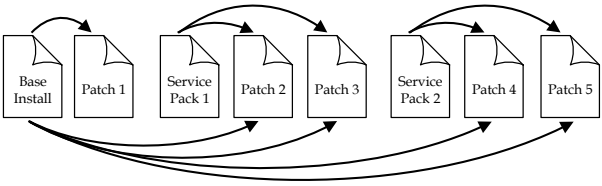
### 3.1 Signature Generation

The basic idea underlying the identification of which vulnerability is exploited by a given document is to use vendor-specific security update information. If program execution starts to behave maliciously at a certain code location and there exists a security-relevant update that, when applied, would change code in the vicinity of that code location, then there is a high probability that one of the vulnerabilities fixed by that update was exploited. As a consequence, we create our signatures from binary differences calculated from security updates.

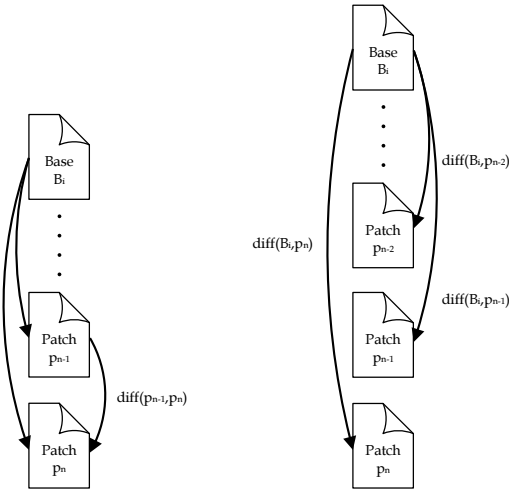
Microsoft offers security relevant updates in so called *msi* setup files. One *msi* file contains the updated versions of the vulnerable files, e.g., if the file *powerpnt.exe* has a vulnerability, the *msi* file will provide the full *powerpnt.exe* and will replace it when applying the update. One complication here for mapping changes introduced by updates to vulnerabilities is the problem of non-original content: an update may duplicate changes of previous (not necessarily security-related) updates so as to allow stand-alone usage, i.e., installation of this update does not require the installation of all previous fixes.

Microsoft also provides major application updates called *Service Packs*. This causes complications when treating updates that build upon a service pack: Firstly, service packs provide functional improvements, bug fixes, and security relevant changes all in one bundle. Security updates that are built upon a service pack necessarily contain large amounts of security-irrelevant code. Secondly, a service pack typically does not update every file of the base installation, yet such files may at a later point of time be changed by an update that requires the service pack. Thus, calculating differences for such updates must take into account both the base installation and the service pack.

The primary goal we want to achieve is to extract the differences between each base patch-level and the original content of each subsequent security-relevant update. The term original in this case means that changes to an installation that have been contained in previous updates but are nevertheless contained in a newer update should be factored out. Based on the previously discussed facts, we need to create a suitable way to identify all security relevant changes of a given update. For updates that are based on the base installation, we simply create the binary difference between the files in the update and the base installation. If an update relies on a service pack, we will create a binary difference between the service pack file version and the updated file, if the file in question is part of the service pack. If that is not the case, we will create the binary difference between the file of the base installation and the updated file of the security update.




(a) The update selection strategy we use for our implementation.




(b) Schematic view on the process to extract $\Delta_{min}$.

**Fig. 2.** Patch selection strategies for the signature generation

Figure 2a illustrates the resulting update selection strategy. In this update selection strategy, we create binary differences between every update and all previous base patch-levels. We use the gathered difference information to calculate the minimal difference $\Delta_{min}$ between two consecutive updates. This is

necessary in order to map changes precisely to the update in which the changes appeared for the first time: the minimal difference will get rid of non-original content. For example, if we consider three consecutive update sets named $p_1$, $p_2$, and $p_3$ and for each of them the binary difference to the same base file $B_i$ has been computed. Then, all changes introduced by update $p_1$ are also included in update $p_2$ and $p_3$ as well and the changes introduced by update $p_2$ are included in the binary differences from the base $B_i$ to update $p_3$. Thus, if we are interested in the changes solely applied by update $p_3$, we need to filter out all changes originating from the updates $p_1$ and $p_2$. In practice this is accomplished by comparing the differences between the base file $B_i$ and update $p_3$ and removing all identical changes found in the differences between the same base file $B_i$ and update $p_1$ and $p_2$ respectively. Note that this step has to be performed on all previous updates which results in a chain of differences of differences.

Figure 2b illustrates the above mentioned process of extracting the minimal difference $\Delta_{min}$ between an update $p_{n-1}$ and $p_n$ which can be formally described as:

$$\Delta_{min} \subseteq \mathit{diff}(B_i, p_n) \setminus \bigcup_{m=1}^{n-1} \mathit{diff}(B_i, p_m) \tag{1}$$

with *diff* being the function to extract the binary differences between the base file and an update and $B_i$ being the $i^{th}$ base file version.

### 3.2 Automatic Exploit Detection and Vulnerability Identification

The last step of the malicious office document analysis process is the dynamic analysis of the document and the identification of the security flaw that is exploited.

In the dynamic analysis we run the malicious document in several sandboxes and monitor the execution using a tool called Pin. The strategy we use to detect an exploit, is based on dynamic detection of corrupted system states by constantly monitoring the process execution. Particularly, we are monitoring each executed instruction. To detect a corrupted system state we defined a rule based on abnormal system behavior. An application is in a corrupted system state, e.g. if the Extended Instruction Pointer (EIP) is pointing to an address outside the code segment.

After detecting a corrupted state, we need to identify the root cause that lead to the application losing control to the malicious code. To achieve this, we are tracing back the execution path that was logged during the EIP observation. Since monitoring happens on a per-instruction basis, we are logging two different traces, one on instruction and one on function level.

In order to achieve the vulnerability identification, we order the list of BBLs, created by Pin, by the order of their occurrence in the execution path. Since Pin logs its information in chronological order, we can determine the BBL that is closest to the point the office application crashed.

In the next step, we identify all those updates from our signature database where the signature is matching the BBL of the previously logged execution and retrieve the information in which update the signature is included. With the help of this information we determine the Microsoft security update including the exploited vulnerabilities referenced by their CVE number.

## 4 Evaluation

In order to demonstrate the feasibility of our approach we examined 7 malicious documents. We captured these documents during several real-world attacks. Our experiments showed that we can reliably detect the misbehavior of the malicious document and further successfully identify, if there is a patch available.

As an initial step, we analyzed each document manually to determine the vulnerability that is exploited and the application version that is vulnerable to the particular attack. Additionally, we used the tools *OffVis* [5], and *officecat* [8] to compare the results. The manual verification of all documents was a very time consuming process and is the main reason that only 7 documents could be evaluated.

Table 1 illustrates the evaluation result of BISSAM when identifying the exploited vulnerability. For 6 out of 7 malicious documents we have correctly identified the security update. Note that one document (CVE_2006_2492.doc) did not trigger the exploit because it was designed for a different Office version, which was not installed in our sandbox environment, and therefore, we were actually able to detect the correct security update for every running malicious document.

Unfortunately, we also detected additional security updates. The reason here is that our approach determines the exploited vulnerability by searching through our signature database and checking whether a BBL matches our trace. As some other updates also patch the same BBL that is listed in our trace, we identify these updates as well. In Section 5 we discuss this problem and give a possible solution. Nevertheless, this problem does not affect the correct detection of the exploited vulnerability and the mitigating security patch.

Table 1 gives an overview of the comparison. In our evaluation, we show that our approach is more reliable in identifying the correct security patch than the approach taken by *OffVis* and *officecat*. Note that both OffVis and officecat are trying to detect the actual vulnerability, referenced by the CVE number, which is particularly harder to determine.

## 5 Limitations and Future Work

Although the evaluation results of BISSAM are already promising, the system still has some limitations. As far as the detection mechanism is concerned, we are limited to the detection of exploits that use techniques implemented in the detection rule, i.e., exploits executed from the stack, or more generally outside the code section. Thus, advanced exploit techniques like *Return Oriented Programming* (ROP) [2,7], currently remain undetected.

**Table 1.** Evaluation results and comparison of the vulnerability patch identification by BISSAM

| Document | Identified Patches by BISSAM | Correct Patch | BISSAM | officecat | OffVis |
|---|---|---|---|---|---|
| CVE_2006_0022.ppt | MS06-028<br>MS06-058 | MS06-028 | ✓ | × | ✓ |
| CVE_2006_2492.doc | | MS06-027 | × | × | × |
| CVE_2009_0556.ppt | MS09-017<br>MS10-004 | MS09-017 | ✓ | × | × |
| CVE_2009_0563.doc | MS09-027<br>MS09-068<br>MS10-036 | MS09-027 | ✓ | × | × |
| CVE_2009_1129.ppt | MS08-051<br>MS09-017 | MS09-017 | ✓ | × | × |
| CVE_2009_3129.xls | MS09-067<br>MS09-021 | MS09-067 | ✓ | × | × |
| CVE_2010_3333msf.doc | MS07-015<br>MS10-087 | MS10-087 | ✓ | × | × |

Further, the detection of BISSAM is limited to systems on which the shellcode triggers. Since the detection routine evaluates the point where control is transferred from the application to the shellcode, the sample needs to be run on the appropriate target system for the exploit to successfully work. Apart from exploits that were designed to work on multiple platform/software combinations, the potentially malicious document would crash the affected software in most cases except the one it is intended to be run on.

When our system tries to identify the update, it relies on the generated signatures and selects all updates that modify anything in the program's execution path. This inevitably leads to the selection of patches that also have non-security relevant changes. As a result, to improve our internal security rating, we evaluate the integration of a security relevance rating following the example of Darun-Grims *Security Implication Score*.

Another limitation concerns the detection of the vulnerability and the identification of the corresponding update. This process fails, if the point of failure in the software is too far away from the point the actual shellcode is executed. In this case the trace misses the executed BBLs that lead to exploitation. Currently, we are tracing the last 5000 BBLs, which according to our evaluation suffices to identify the exploited vulnerabilities correctly.

## 6 Conclusion

In this paper, we improve the analysis of malicious documents by presenting a novel approach called BISSAM to automatically identify the exploited vulnerability and detect the embedded shellcode. Our system consists of two parts: The first part is responsible for creating signatures based on vendor's update information in an automated fashion. The second part comprises the actual analysis system, which consists of several sandboxes running different office application versions and patch-levels. Each document is opened in every application-specific

sandbox and our *pintool* traces the execution. After the document has been executed in every sandbox, the trace is evaluated using the previously generated signatures. The result is a list of possible vulnerabilities that the document exploits.

The evaluation results showed that our approach is not only able to reliably detect malicious documents, but also to extract the involved shellcode, and to identify the exploited vulnerability, i.e., we are able to determine the update that fixes the corresponding security issue.

In summary, the presented approach performs well on current threats concerning office documents and forms a great addition to today's security measures in the field of client application attacks.

**Acknowlegement.** We would like to thank Heiko Patzlaff for the various discussions about the exploit detection techniques and his feedback on an early version of our work. We are indebted to Bernd Grobauer, Tobias Limmer, and Felix Freiling for their comments on this paper.

## References

1. Boldewin, F.: OfficeMalScanner (2011), `http://www.reconstructer.org/code.html`
2. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: Generalizing return-oriented programming to RISC. In: Syverson, P., Jha, S. (eds.) Proceedings of CCS 2008, pp. 27–38. ACM Press (October 2008)
3. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of ACM SIGPLAN Conf. Programming Language Design and Implementation, PLDI 2005 (June 2005)
4. Microsoft: Microsoft Security Intelligence Report (SIR), vol. 7 (January-June 2009)
5. Microsoft: OffVis 1.1 (2009), `http://blogs.technet.com/b/srd/archive/2009/09/14/offvis-updated-office-file-format-training-video-created.aspx`
6. Oh, J.: DarunGrim: A Patch Analysis and Binary Diffing Tool (2011), `http://www.darungrim.org`
7. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: De Capitani di Vimercati, S., Syverson, P. (eds.) Proceedings of CCS 2007, pp. 552–561. ACM Press (October 2007)
8. Snort Project: OfficeCat (2010), `http://www.snort.org/vrt/vrt-resources/officecat`

# Self-organized Collaboration of Distributed IDS Sensors

Karel Bartos[1], Martin Rehak[1,2], and Michal Svoboda[2]

[1] Faculty of Electrical Engineering
Czech Technical University in Prague
Prague, Czech Republic
{karel.bartos,martin.rehak}@agents.fel.cvut.cz
[2] Cognitive-Security s.r.o., Prague, Czech Republic

**Abstract.** We present a distributed self-organized model for collaboration of multiple heterogeneous IDS sensors. The distributed model is based on a game-theoretical approach that optimizes behavior of each IDS sensor with respect to other sensors in highly dynamic environments. We propose a general formalization of the problem of distributed collaboration as a game between defenders and attackers and introduce $\varepsilon$-FIRE, a solution concept suitable for solving this game in highly dynamic environments.

Our experimental evaluation of the proposed collaboration model on real network traffic clearly shows improvements in the detection capabilities of all IDS sensors, allowing each system to specialize on particular network activities while not reducing the overall effectiveness. The concept of opponent aware, self-coordinating and strategically reasoning Network Intrusion Detection Networks allows effective collaboration of individual system defenders that may match a market-based collaboration structures of the attackers.

## 1 Introduction

Protecting network security assets against modern, highly sophisticated network attacks represents a complex challenge for researchers and security experts. Lots of successful targeted attacks have shown large vulnerabilities and unpreparedness of corporate network security mechanisms to face novel and more advanced network threats. Intrusion Detection Systems (IDS) are a widely used mechanism for network protection, which helps to secure network infrastructures by using static signature matching or dynamic anomaly detection methods. Signature-based IDS systems evaluate each network connection according to predefined signatures regardless of the context, showing promising results on well-known attacks, but with limited capabilities to detect novel intrusions. On the other side, anomaly-based IDS systems are designed to detect a wide range of network anomalies including yet undiscovered attacks, but at the expense of higher false alarm rates. Thus each sensor perceives information differently depending on its functionality or deployment. Considering the benefits and limitations of each,

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 214–231, 2013.

the key to detect nowadays advanced threats and collaborative attacks lies in a distributed collaborative mechanism consisting of multiple heterogeneous IDS systems deployed in various parts of the network infrastructure.

The proposed research work is specifically aimed at the investigation of global distributed collaboration of individual autonomous detectors and the relationship between system response characteristics (e.g. detection sensitivity), stability of these characteristics, and their predictability by the opponent. We define a game-theoretical framework suitable for the collaboration of multiple heterogeneous IDS systems and introduce simple yet effective game solution concept $\varepsilon$-FIRE, where multiple IDS sensors seek to optimize global collective objective through local decision making. The $\varepsilon$-FIRE concept combines $\varepsilon$-greedy method with FIRE model to maintain robust and efficient properties suitable for dynamic environments.

We propose to specialize each IDS sensor to detect unique intrusions and attacks that have not been detected elsewhere. More specialized IDS sensors may produce less true and false positives, which can decrease individual effectiveness of the sensors. However, specialized sensors provide more precise and valuable alerts, because these alerts are unique and are provided with higher confidence. Thus, the proposed collaboration of more specialized IDS sensors, where each sensor dynamically reconfigure its parameters and focuses on specific types of intrusions, results in better overall detection coverage of the attacks. We have evaluated the proposed concept on academic networks, where we have shown considerable improvements of the $\varepsilon$-FIRE collaboration in detection capabilities of intrusion detection devices.

The paper is structured as follows. In Section 2 we discuss general assumptions of a distributed collaboration. Formalization of the collaboration with the game-theoretical approach as an extensive game between defenders and attackers is proposed in Section 3, while Section 4 contains a discussion of existing solution concepts for solving the game-theoretical problems. In Section 5, we introduce $\varepsilon$-FIRE, the game solving concept that can be applied to solve the game in highly dynamic environments. Our experimental evaluation of $\varepsilon$-FIRE concept is described in Section 6 and Section 7 concludes the paper with practical implications of our findings and a summary of the results.

## 2 Distributed Collaboration

In a collaborative IDS system, each node shares information with other nodes according to the predefined policies, allowing the node to adapt both locally and globally with respect to received information. Sharing information among multiple heterogeneous systems can be also utilized by fusion methods [5] to reduce false alarm rates or find some relations among the reported alerts. Moreover, results from various parts of the network infrastructure may reveal more complex attack scenarios. But the distributed information sharing can be also used for a collaborative co-adaptation, where each node adapts on the network environment with respect to the results provided from other nodes, which we

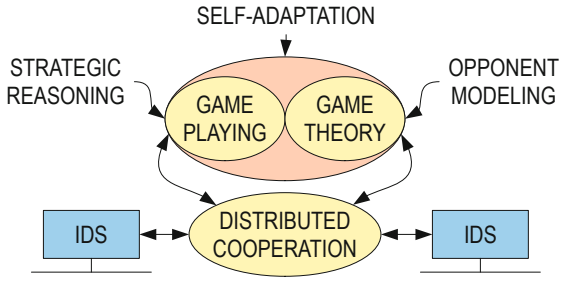



**Fig. 1.** General game-theoretical model for a distributed collaboration and high-level assessments of multiple heterogeneous IDS sensors

believe is the key property when detecting advanced collaborative attacks. General game-theoretical approach for a distributed cooperation is depicted in Fig. 1.

The problem of distributed collaboration has been studied by the research community from two perspectives. The majority of research focuses on multi-sensor alert correlation and data fusion techniques, where the goal is to fuse alerts from heterogeneous sensors to provide more reliable output of the system, e.g. by reducing false alerts [5]. However, another possible approach is to employ alert correlation into a feedback mechanism that would influence the behavior of all nodes and the whole collaborative system would react as an intelligent and robust Intrusion Detection Network.

The proposed collaboration model uses alert correlation in two ways: first it is used for accumulating collective information from the outside of the node (typically for accumulating results from other nodes), allowing each node to adapt on the current network state from the global point of view. An example of such adaptation is change in parameters or threshold values, modification of inner models, change in priorities of rules, adding new patterns or signatures etc. Moreover, alert correlation is also used for summarizing the alerts provided by all collaborating nodes to create global network security awareness.

In our work, we assume that the monitored network is covered by multiple heterogeneous IDS systems (nodes). These heterogeneous IDS nodes detect attacks and intrusions by using various detection mechanisms and types of input data - netflows, signatures, logs, etc. We introduce a game-theoretical framework for a distributed co-adaptation that requires the following assumptions:

- **Local self-monitoring** - all IDS nodes should be able of a local reconfiguration to adapt on the current state of the network according to the proposed game model.
- **Interoperability** - outputs of all nodes should be in the standardized format (e.g. Intrusion Detection Message Exchange Format - IDMEF [4]), allowing their interaction even if their detection mechanisms are different. We will refer to these outputs as *events*.
- **Communication** - maintaining robust and reliable communication among multiple IDS nodes is essential assumption in the distributed collaboration. We will discuss this aspect further in this section more in detail.

- **Security** - for security reasons, nodes do not provide information about their internal state. Furthermore, secure communication channel should be provided to reduce the possibility of attacker's manipulation with the system.
- **Traffic assumptions** - strategic deployment of IDS nodes in the network is important to provide relevant information to the game model.

The above-mentioned assumptions define the initial conditions of the proposed distributed co-adaptation controlled by a game-theoretical model explained further in Section 3.

### 2.1 Communication

An important aspect of the distributed collaboration is the communication protocol. In our model, each node can communicate with the rest of the nodes to be able to provide results in a fully distributed manner. We justify the choice of the distributed topology for its scalability and security properties - it does not introduce a single point of failure. Moreover, possible communication overhead can be reduced by grouping the alerts from a single node into one message, which can be periodically sent to other nodes.

Maintaining situational awareness in distributed groups represents a difficult task as well. The communication protocol should be as light as possible, with no synchronization issues. An example of such light-weight protocol is publish-subscribe protocol, where at the beginning a node informs others that it provides and requires results in form of alerts. After this subscription, this node will send alerts to the subscribed nodes and at the same time will expect remote alerts generated by other nodes. Once all alerts are received (or timeout elapses in case of some failure), nodes may perform further collaboration processing having all available alerts at disposal.

## 3 Game-Theoretical Model

We formalize the distributed co-adaptation as a game between attackers and a set of defenders represented by individual IDS nodes. Each player performs certain actions to achieve its predefined goal. An example of attacker's goal is to exploit some secret data from private network. On the other hand, defenders' goal is typically to prevent attackers from achieving their goals.

More formally, the **defender** system consists of $n$ IDS detectors $P_D^1, \ldots, P_D^n$. Each IDS detector is one player of the game with its own set of strategies. We will denote the set of all strategies for player $P_D^j$ as $X_D^j$, and $i$-th strategy of player $P_D^j$ as ${}^i x_D^j$. Similarly, the set of **attackers** $P_A^1, \ldots, P_A^m$ uses strategies $x_A^1, \ldots, x_A^m$ from $X_A^1, \ldots, X_A^m$. By using this notation, we define the game of $n$ defenders and $m$ attackers as follows:

$$G = (P, X, U) \tag{1}$$

$$P = \{P_A^1, \ldots, P_A^m, P_D^1, \ldots, P_D^n\} \tag{2}$$

$$X = \{X_A^1, \ldots, X_A^m, X_D^1, \ldots, X_D^n\} \tag{3}$$
$$U = \{u_A^1, \ldots, u_A^m, u_D^1, \ldots, u_D^n\} \tag{4}$$
$$u_A^i \ : \ X_A^i \times X_D^1 \times \ldots \times X_D^n \to \mathbb{R} \tag{5}$$
$$u_D^j \ : \ X_A^1 \times \ldots \times X_A^m \times X_D^j \to \mathbb{R}. \tag{6}$$

As you can see, utility function of $i$-th attacker $u_A^i$ depends on his strategy and strategies of all IDS nodes, while utility function $u_D^j$ of $j$-th defender depends on strategies of the defender and all attackers. The attacker and defender system should ideally optimize a shared utility function. In practice, though, the utility functions (on the defender's side) differ as the internal state of each player is inaccessible for integration and security reasons. The dynamism of the environment causes that utility functions of the defender can rarely be identical. This diversity in utility functions increases robustness and wide adaptability of the overall defender system.

In the following, we will assume a three-player game between one attacker and two defenders. Note that this game can be simply extended to different configurations with more players on both sides. The payoff matrix for the first defender that describes payoff distribution according to strategies selected by the attacker (columns) and the first defender (rows), can be written as follows:

$$\mathbb{U}_D^1 = \begin{pmatrix} u_D^1({}^1x_A^1, {}^1x_D^1) & \ldots & u_D^1({}^kx_A^1, {}^1x_D^1) \\ \vdots & \vdots & \vdots \\ u_D^1({}^1x_A^1, {}^lx_D^1) & \ldots & u_D^1({}^kx_A^1, {}^lx_D^1) \end{pmatrix}, \tag{7}$$

where $k$, $l$ denotes the number of all strategies for $P_A^1$, $P_D^1$ respectively. $I$-th column of matrix $\mathbb{U}_D^1$ describes the payoff changes when attacker selects strategy ${}^ix_A^1$ depending on defender strategies in rows. The payoff matrix for the attacker $\mathbb{U}_A^1$ and the second defender $\mathbb{U}_D^2$ can be expressed similarly.

Until now, we have not introduced any collaboration link among players and the game can be solved for each player locally with no other interactions. However, we extend this model with a collaboration mechanism, where defenders share their results (not internal states) and on these bases each defender modifies the payoff matrix to adapt the system in a distributed manner. The presented distributed collaboration consists of three main steps:

1. **sharing distributed results** - in the first phase, each defender collects results from all cooperating defenders for the given time period (e.g. alerts from 5 minutes of network traffic). We will denote the results of $i$-th defender as $R_D^i$.
2. **computing feedback matrix** - based on all results from the given time period, $i$-th defender computes the feedback matrix $\mathbb{F}_D^i$. In our three-player game, the feedback matrix of the first defender will be as follows:

$$\mathbb{F}_D^1 = \begin{pmatrix} f_1({}^1x_A^1, {}^1x_D^1, R_D^1, R_D^2) & \ldots & f_1({}^kx_A^1, {}^1x_D^1, R_D^1, R_D^2) \\ \vdots & \vdots & \vdots \\ f_1({}^1x_A^1, {}^lx_D^1, R_D^1, R_D^2) & \ldots & f_1({}^kx_A^1, {}^lx_D^1, R_D^1, R_D^2) \end{pmatrix}, \tag{8}$$

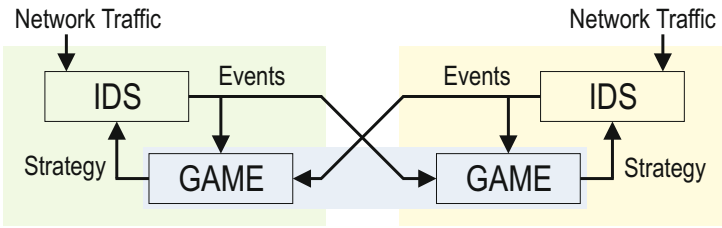



**Fig. 2.** The proposed game-theoretical scheme for the distributed collaboration of multiple heterogeneous IDS sensors

where $f_i$ is the *feedback function* that evaluates results of used strategies depending on results from other nodes and transforms them into real values. This feedback function tells us relative quality of the system for given strategies and results.

3. **altering internal state** - once the feedback matrix is computed, each defender creates the final payoff matrix ${}_F\mathbb{U}$ as a weighted average of local and remote utility values:

$$ {}_F\mathbb{U}_D^1 := w_u \cdot \mathbb{U}_D^1 + w_f \cdot \mathbb{F}_D^1, \qquad w_u + w_f = 1. \tag{9}$$

To create the final payoff matrix, we do not need to know attacker's payoff, because ${}_F\mathbb{U}$ depends only on payoff matrices and results of the defenders. Such modification of a payoff matrix typically results in a change of the optimal strategy and thus each defender optimizes its performance with respect to other defenders to maximize collective gain.

The notion of independent optimization, where players are not explicitly informed about actions played by other players, makes the game consistent with the formalization based on extensive-form games introduced in [15] and discussed in Section 4. We are facing a dynamic optimization problem, where the environmental conditions imposed by the external environment can change rapidly, making the game similar to a sequence of static games with unpredictable length.

## 4 Current Solution Concepts

Algorithms and solution concepts have been widely studied and formalized in static environments, where the repeated game converges to Nash equilibrium (i.e. stable point in the strategy space where none of the players benefits from unilateral deviation). However, in highly dynamic environments (like computer networks), behavior of the optimal algorithm is subject of recent research and has many unknown properties that are not yet well described. Pareto-optimal algorithms converge to more equilibria that may change in time as the game conditions evolve, so the system should be flexible and scalable for changing policies and goals. That means the optimal algorithm should select among more equilibria rather than converge to a single one.

Regret minimization is a technique achieving long-term optimality in a sequence of identical games. It does not require any knowledge of other players' utility functions, but is based on independent loss minimization performed by individual players given the payoffs received for their past moves – the strategy played by the agents is strictly based only on the feedback that they receive. The regret, or more specifically the *external regret*, is defined [3,6] as an ex-post evaluated *loss of utility* due to the suboptimal strategy selection - thus the term regret.

Regret minimization is a robust method that yields predictable results in a wide range of games. In a static environment, it can be shown that the use of the proper regret minimization algorithms bounds the maximal loss achieved using external regret minimization by the term $O(\sqrt{T \log |X|})$ relative to the best loss achievable [3]. This is a general result that can be applied outside of game theoretic frameworks.

In the specific case of two-player, zero-sum games, the use of regret minimization will make the player's payoffs converge towards the value of the game, with the speed of convergence bounded by the term above. This result can be generalized for a far broader class of games. Hart and Mas-Colell show that if *all* players play regret minimization in a sequence of static games, the joint distribution of play converges [8,6] to the set of *correlated equilibria* [10,2] of the stage game. One of the important corollaries is that the probability of strategy switching decreases as well, making the players reach increasingly longer sequences of constant strategy play.

The correlated equilibrium is an extension of the well-known Nash equilibrium by assuming that the players can either communicate, or can observe a common variable(s), or share a history of gameplay. All these specific examples are special cases of a *correlating device* [3]. Such a device produces a set of (correlated) signals, one for each player, which use for strategy selection in the game. It can be shown that when the signals are fully correlated (e.g. when all players share a single public signal), the correlated equilibria set equals the convex hull of Nash equilibria.

On the other hand, convergence to the smaller set of Nash equilibria is possible, but is guaranteed only in very specific types of game, and not guaranteed at all in the general case [7]. In particular, in the two player, zero sum game example mentioned above, the game converges, but counter-examples of non-convergent games can be found even for simple three player games, such as the Shapley game [12]. In non-zero sum games with more than two players, the regret-minimizing algorithm provides robust results when other approaches may fail.

The results of [15] suggest that regret minimization is also robust in zero-sum finite extensive-form games with perfect recall when applied across independent information sets in the game. The regret (counterfactual regret) is measured and minimized on information sets in the game and the authors show that minimization of the counterfactual regret in individual game stages bounds the overall regret of the global game – albeit only in a very specific class of games.

This is an extremely important property, as it hints that at least some of the regret minimization properties may hold when we split one of the players into several partial players, each of them selecting a fraction of the original player's strategy.

The question of convergence is further complicated in the dynamic environment. The algorithm's convergence speed becomes critical, as it needs to converge within the time interval when the environment (which defines the structure of the game) is relatively stable, making the set of correlated equilibria also stable. This also implies that the value of the past history decreases with increasing time difference to find the balance between the robustness and speed of convergence.

## 5 The $\varepsilon$-FIRE Algorithm

In this section, we will introduce a simple, yet effective algorithm that can be used as a solving concept for the game defined in Section 3. As mentioned above, algorithms in dynamic network environment should converge fast into equilibrium that can be easily changed with the change of the environment. The $\varepsilon$-FIRE algorithm combines FIRE model [9] with $\varepsilon$-greedy [13] method to guarantee that the algorithm will always find new equilibria.

The $\varepsilon$-greedy algorithm is an effective means of balancing exploitation and exploration in multi-agent reinforcement learning domain. This algorithm behaves greedily most of the time, which means that it selects the actions with the highest estimated reward (or rating). However every once in a while (with probability $\varepsilon$), it selects the action randomly from all possible actions:

1. Algorithm starts with a set of strategies $X$ to select from. Each strategy is associated with expected utility $u(x)$.
2. The algorithm draws a random number $r$ from the $[0, 1]$ interval. It compares $r$ with the $\varepsilon$ parameter (hence $\varepsilon$-greedy).
3. If $r < \varepsilon$, then the algorithm randomly selects one strategy from the strategy set. Otherwise, it selects the strategy with the highest expected payoff.

The convergence of the $\varepsilon$-greedy algorithm is guaranteed [13], because as the number of selections increases, all actions will be selected an infinite number of times.

The results regarding the behavior of the $\varepsilon$-greedy algorithm in stochastic games are encouraging. In [14], the authors show that the application of $\varepsilon$-greedy can achieve better results than standard Q-learning approaches in a series of games. Interestingly, the authors argue that its use can achieve an average payoff higher than Nash equilibria value in some games and show it for Prisoner's dilemma in particular. However, compared to regret-minimization algorithms [3] discussed in Section 4, the algorithm does not use (and it also does not need to maintain) the information about the expected payoff for each action.

FIRE model [9] was originally designed for trust evaluation in open multi-agent systems. We have adopted its interaction part into our algorithm to

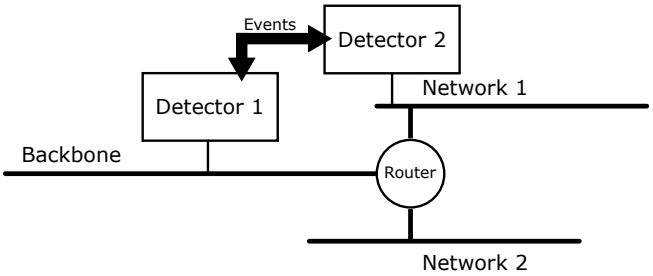



**Fig. 3.** Collaboration scenario between two IDS nodes deployed on different parts of the network. The first IDS was placed topologically in front of the second IDS.

increase convergence speed. Greedy strategy with the highest payoff is computed according to the following equation:

$$u(x) = \max_j \sum_i w_i \cdot u_i^j, \tag{10}$$

where $w_i$ represents $i$-th weight coefficient and $u_i^j$ represents $i$-th last observed payoff of $j$-th strategy. The weights must hold conditions:

$$\sum_i w_i = 1 \quad \wedge \quad w_{i-1} \leq w_i, \tag{11}$$

which means it gives more weight to more recent payoff. Typically, we define FIRE model with five past observations. Thus the $\varepsilon$-FIRE algorithm selects strategy $x$ according to the following principle:

$$x = \begin{cases} \arg\max_j \sum_i w_i \cdot u_i^j & r \geq \varepsilon \\ \\ \text{random}(X) & r < \varepsilon \end{cases} \tag{12}$$

The $\varepsilon$-FIRE algorithm is suitable to use in highly dynamic environments, where the reward variance is larger and rewards are noisier. In such environments, $\varepsilon$-greedy method should perform better than any simple greedy algorithm [13]. We also argue that in highly dynamic environments, the $\varepsilon$-FIRE algorithm can outperform the regret minimization as the regret values based on long-term past experience may be misleading.

## 6 Experimental Evaluation

In this section, we will describe the results of our distributed experimental scenarios, incorporating a distributed collaboration of two IDS nodes deployed on different parts of the network infrastructure (as shown in Fig. 3). The first node (denoted as *backbone*) processes traffic from the backbone network, while the second node (denoted as *subnet*) processes traffic acquired only from a part of

the whole network (it does not see traffic between the backbone and network 2). Both nodes are influenced by each other's events and therefore, their results will diverge as they produce different events and make different decisions.

In all scenarios, we used the same datasets consisting of 500 minutes of university network traffic, which we have partially classified. Furthermore, these datasets contain persistent real malware traffic. We have established an informal collaboration with a penetration tester who did supply us with a sample of sanitized malware used for penetration testing of enterprise customers based on social engineering. The malware behavior consists of various stages, including the initial infection (typically from a USB drive or an email attachment), (optional) library download through the HTTP connection and principally the Command and Control traffic implemented as a periodic polling of a specific website with HTTP GET requests modeled after actual malware behavior. The script establishes and maintains the connection and downloads a small file from the Command and Control server as well.

### 6.1 IDS Nodes Description

For our evaluation, we used two CAMNEP [11] IDS nodes. Each node analyzes 5-minute blocks of incoming and outgoing network traffic in the netflow [1] format by using six different anomaly detection methods. Each anomaly detection method uses its own detection algorithm and assigns to each flow corresponding marginal degree of anomaly. Once all detection methods finish their processing, CAMNEP system chooses suitable combination of the detection methods and evaluates each flow with final degree of anomaly, which is computed as a weighted average of all marginal degrees of anomaly. The system optimizes its effectiveness by selecting suitable combination of the methods that would detect most of intrusions.

To find the best combination of the detectors, the system uses local adaptation process called *challenge insertion*. It continuously inserts predefined legitimate and malicious network traffic (challenges) from the database into the real network traffic and evaluates the performance of the detectors on these challenges. The combination of the detectors, which performs best on challenges, is selected for final aggregation on real network traffic.

CAMNEP IDS node has the following properties:

- **online detection** - CAMNEP system processes datasets with 5 minutes of network traffic (because of statistical nature of anomaly detection methods).
- **local self-adaptation** - CAMNEP system satisfies our assumption of self-monitoring mentioned in Section 2, because the system is able to reconfigure and change its internal state. This reconfiguration is defined as a change in the combination of the detection methods used for the final aggregation of current anomaly assessments. One way of controlling this reconfiguration is through local adaptation process called challenge insertion.
- **threshold computing** - based on self-adaptation, the system determines the threshold dividing legitimate and malicious parts of network traffic.

Once final degree of anomaly is assigned to all flows, a clustering mechanism clusters anomalous flows with similar characteristics into events characterizing anomalous processes and services in the network. Different aggregation function (defining the combination of the detectors) typically results in different events. More detailed description of CAMNEP system can be found in [11].

In our experimental evaluation, the suitable combination of the detectors is controlled by the local self-adaptation (challenge insertion), or by the proposed collaborative framework ($\varepsilon$-FIRE). Each CAMNEP system has 30 different aggregation functions to choose from.

### 6.2 Collaborative Game Settings

As mentioned above, our game has two defenders represented by IDS nodes and one attacker controlling malware activity. The goal of the defenders is to detect malicious activities on the network (including malware). To achieve this goal, IDS node may use different strategies represented by selecting suitable aggregation function of the detectors within a node. The optimal strategy is a combination of detectors where malicious activities are successfully detected. We assume that the optimal strategy may change with dynamic changes of the network environment.

Each IDS node updates payoff matrix $\mathbb{U}$ defined in Eq. 7 based on local self-monitoring (challenge insertion - see Section 6.1). In collaboration scenario, each node also considers the knowledge gained from other nodes that is represented in feedback matrix $\mathbb{F}$ - see Eq. 8. As a feedback function, we have chosen *uniqueness* of local events when compared to events received from other nodes. Algorithm of computing uniqueness is described in Alg. 1.

This game setting allows each node to model the suitability of aggregation functions not only w.r.t. challenge insertion results (as local self-adaptation), but also w.r.t. the *uniqueness* of created events. Thus each system node is encouraged to specialize on unique network behaviors while still performing well in known

**Algorithm 1 .** Uniqueness computation

```
function computeUniqueness(localEvents, remoteEvents)
    unique = 0
    for localEvent : localEvents do
        for remoteEvent : remoteEvents do
            unique = unique + 1 − eventSimilarity(localEvent, remoteEvent)
        end for
    end for
    return uniqueness
end function

function eventSimilarity(lE, rE)
    return 2 ∗ intersection(lE.flows, rE.flows)/(lE.flows + rE.flows)
end function
```

situations represented by challenges. In our experiments, we use the malware traffic detection to assess the success of specialization.

### 6.3 Experimental Results

In our evaluation, we will analyze malware detection effectiveness of IDS node running in stand-alone configuration without any interventions from other systems (denoted as *stand-alone*) and compare this baseline configuration with results from collaborative scenario illustrated in Fig. 3 with three collaboration strategies: *$\varepsilon$-greedy E*, *$\varepsilon$-greedy CH*, and *no-feedback* strategy.

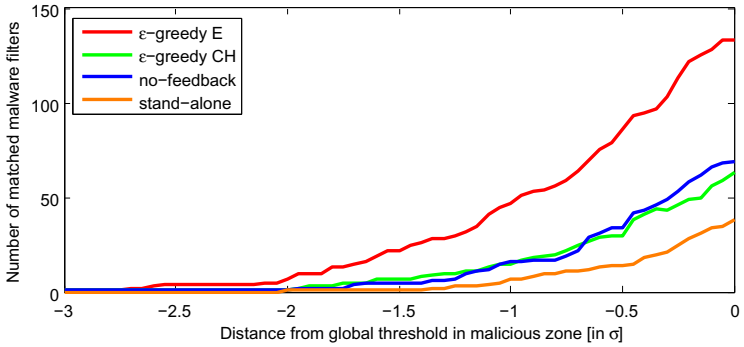
In $\varepsilon$-greedy E, the system node uses $\varepsilon$-FIRE algorithm introduced in Section 5 to select optimal aggregation function. We have defined FIRE model (see Section 5) with vector of weights $w = \{0.046, 0.082, 0.147, 0.261, 0.464\}$ (exponential decrease) and set game weights introduced in Eq. 9 as $w_u = 0.3, w_f = 0.7$ (which means weight 0.3 on local self-adaptation and 0.7 on distributed feedback – uniqueness of events), thus encouraging mutual specialization. We enabled dynamic exploring in the space of aggregation functions by setting the exploration rate of $\varepsilon$-FIRE algorithm as $\varepsilon = 0.2$ and using initial optimistic values technique to boost early exploration in the space of aggregation functions.

Strategy denoted as *$\varepsilon$-greedy CH* uses $\varepsilon$-greedy algorithm as well, but with different weights: $w_u = 1.0, w_f = 0.0$, and $\varepsilon = 0.2$. This strategy can be considered as direct extension of local self-adaptation (also based on FIRE model) with exploring possibilities.

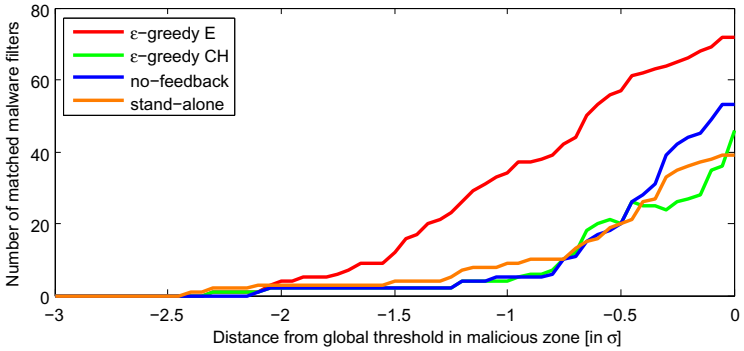
Finally, the last distributed strategy *no-feedback* is very simple, because it includes default stand-alone setting enriched with events fusion technique, where the system nodes share and fuse events between each other without any feedback for selecting optimal aggregation function. The fusion is very simple process, where each node includes unique remote events into its own set of output events.

In the following figures, we will show how distributed collaboration techniques improves detection capabilities of individual IDS systems. In Fig. 4 we can see how many times the system successfully detected malware as malicious activity by placing the traffic into the malicious zone (below the threshold). We consider malware to be detected if and only if the system creates an event with malware traffic. Note that better separation from the threshold (in sigma distance) leads to more reliable detection. Sigma distance of an event is defined as a difference between the threshold position and average degree of anomaly of all flows in the event (computed by IDS node) in standard deviation (computed from anomaly values of all flows). Higher negative values means better separation from the threshold in the malicious zone.

Analysis from subnet IDS node is depicted in Fig. 4 (a), where we can clearly see the benefits of $\varepsilon$-FIRE algorithm ($\varepsilon$-greedy E), when both systems interact and reconfigure. The number of successfully detected malware traffic was doubled when compared to the case when both systems only combine and fuse their results (no-feedback) or adapt based on local information with exploration possibilities ($\varepsilon$-greedy CH), which is still much better than stand-alone configuration, when both systems ran separately. Note that the evaluation datasets




(a)

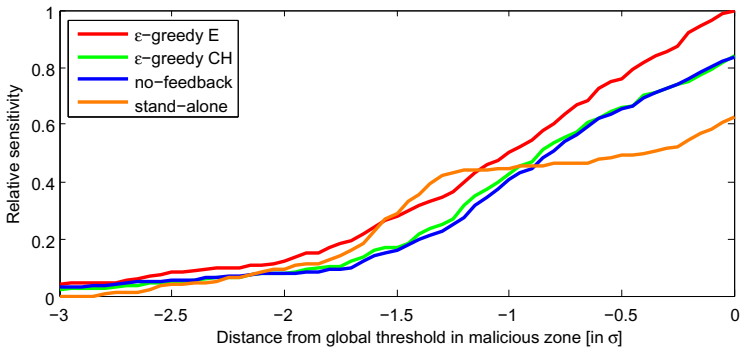



(b)

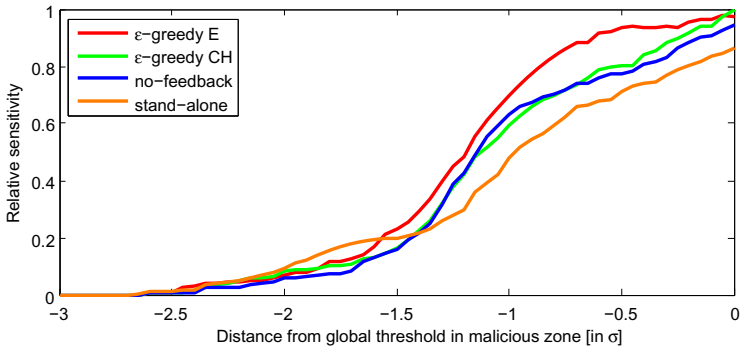
**Fig. 4.** Number of successfully detected malware traffic depending on sigma distance from the threshold position - subnet (a) and backbone (b) location. Higher values are better.

contain 200 malicious events with the malware traffic (malware requests and responses during 100 5-minute intervals), which means that the proposed collaboration technique increased the amount of successfully detected malware (i.e. the number of malware events below the threshold, $\sigma = 0$) from 23% to 70%.

We performed analogous analysis on backbone IDS node as illustrated in Fig. 4 (b). The results are similar as in the subnet IDS node, however the difference between $\varepsilon$-greedy E ($\varepsilon$-FIRE) and other techniques is not so significant in number of detected malware. No-feedback (fusion) configuration as well as $\varepsilon$-greedy CH shows comparable results with stand-alone configuration in number of found malware (Fig. 4 (a)). This suggests that event fusion technique is not sufficient. Therefore the system requires collaborative approach which ensures system diversity and specialization as demonstrated in Fig. 4 with $\varepsilon$-greedy E feedback strategy. This is especially important given the fact that the self-monitoring mechanism on both nodes was identical, and that the improved behavior is purely a result of co-specialization.




(a)



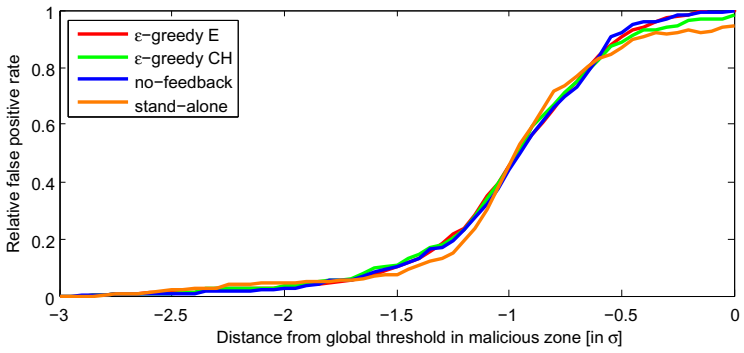

(b)

**Fig. 5.** Relative detection sensitivity of the system depending on a sigma distance from the threshold - subnet (a) and backbone (b) location. Higher values are better.

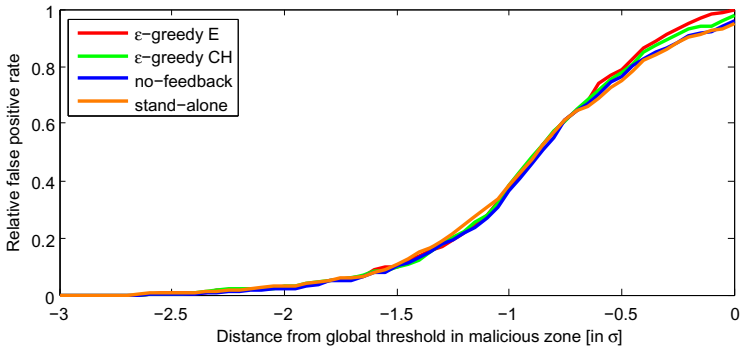
You can see that backbone IDS node detected less malware events than subnet IDS node. It is because of the fact, that subnet IDS node analyses less traffic, allowing the node to maintain more detailed models of underlying network state.

In order to verify the benefits of distributed collaboration, we have to analyze not only detection effectiveness on malware traffic, but also all malicious and legitimate traffic. For this reason, we have compared the performance of each configuration w.r.t. true and false positives ($TP$ and $FP$). True positives correspond to created events with malicious or suspicious behavior, and on the contrary, false positives correspond to created events matching legitimate behavior. To better express the differences of the selection strategies, we put all metrics on relative scale normalizing by the highest achieved value. By using such scaling, we can compare each method relatively easily.

We have decided to use relative values to show the differences between the stand-alone architecture and the proposed collaborative design. We believe that absolute values depend on the type IDS system used for the evaluation and therefore are not so informative as relative differences.




(a)

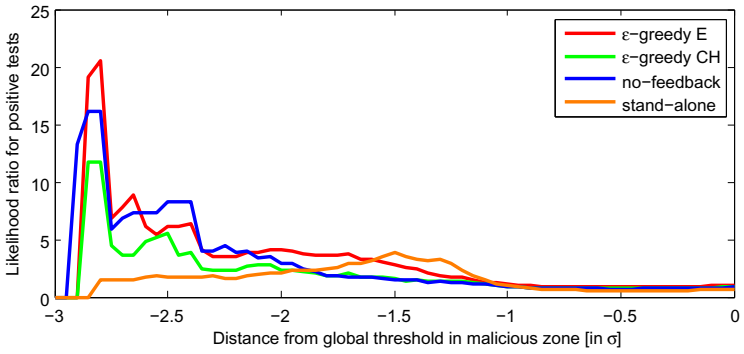



(b)

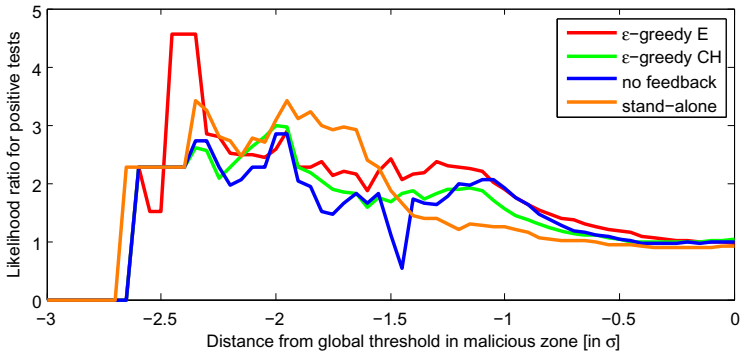
**Fig. 6.** Relative false positive rate of the system depending on a sigma distance from the threshold - subnet (a) and backbone (b) location. Lower values are better.

Relative detection sensitivity of both IDS nodes (illustrated in Fig. 5) is computed as $TP/(TP+FN)$, where $TP$ denotes the number of successfully detected malicious events and $TP + FN$ denotes the number of all malicious events. We can see that in subnet node (Fig. 5(a)) $\varepsilon$-greedy E detects the most of malicious traffic, however around $1.3\sigma$ from the threshold it is outperformed by stand-alone system showing the peak sensitivity. Still we can say that $\varepsilon$-greedy CH and no-feedback detected 80% and stand-alone configuration 60% of the malicious traffic when compared to $\varepsilon$-greedy E. Note that the absolute number corresponds to the maximal relative sensitivity is 0.71.

The results of backbone IDS node (see Fig. 5(b)) show very similar results as the subnet IDS node, having smaller differences in detection sensitivity. We believe it is due to the fact that subnet IDS node is more influenced by backbone IDS node allowing to alter its original standard behavior. Backbone system is able to use the subnet specialization to enhance its own detection sensitivity to address other threats instead. The absolute number corresponds to the maximal relative sensitivity in the backbone location is 0.52.




(a)




(b)

**Fig. 7.** Likelihood ratio for positive test of the system depending on a sigma distance from the threshold - subnet (a) and backbone (b) location.

Furthermore it is interesting that all configurations show comparable results of false positive rates (Fig. 6). This shows that the specialization does not necessarily induce locally sub-optimal behavior. Absolute false positive rate in the subnet location was 0.29, while in the backbone location was 0.34. However, a lot of true negatives were not included into these evaluations, because of the fact that the network traffic was not labeled completely. However, for a demonstration of the benefits of the proposed collaborative approach, we believe the relative differences as depicted in Fig. 5 and Fig. 6 are representative enough.

Finally, we have compared the tradeoff between detection sensitivity and false positive rate by using likelihood ratio for positive test, as illustrated in Fig. 7:

$$\text{likelihood ratio for positive test} = \frac{\text{sensitivity}}{\text{false positive rate}} = \frac{TP/(TP+FN)}{FP/(FP+TN)}.$$

The higher the value, the higher the probability that an event corresponds to true positive. Stand-alone subnet IDS node (Fig. 7 (a)) has much lower likelihood ratio when compared to the rest of the techniques. However, we can assess that

the most effective distance ends $1.5 - 1\sigma$ from the threshold position - both for subnet and for backbone system node.

In this section, we have performed various evaluations of stand-alone system running individually and have compared it with three collaborative strategies, where this system is communicating with another system node located in different part of network infrastructure. We have demonstrated that event fusion from collaborative CAMNEP nodes has clear positive impact on the overall detection performance. Specifically, it increases detection sensitivity while false positive rate remains constant. Moreover, we have clearly shown that proposed distributed co-adaptation between nodes can significantly extend the detection potential, allowing each system to specialize on particular network activities while not reducing the overall effectiveness. By deploying distributed collaborative approaches to more complex network infrastructure, the system nodes will dynamically react to the changes in the network environment. They are able to act as a coherent entity capable of maximizing global network security awareness, while relying only on minimal mutual collaboration and implicit synchronization.

## 7 Conclusion

The proposed work aims to shift from individual, local intrusion detectors to the robust global security mechanism covering whole network infrastructure. The proposed distributed architecture benefits from collective information sharing, where all individual detectors contribute to global modeling of the underlying network state, while strategically selecting the optimal amount of information to share. Moreover, each detector shall be able to adaptively modify its own local model on the basis of globally coordinated game playing strategy against corresponding opponent's sophisticated scenario.

We have proposed game-theoretical framework of distributed collaboration and $\varepsilon$-FIRE concept for solving this game in highly dynamic network environments. From our experimental evaluation, we have clearly shown improvements of the multi-sensor collaboration controlled by $\varepsilon$-FIRE technique, allowing each system to specialize on particular network activities while not reducing the overall effectiveness. The concept of opponent aware, self-coordinating and strategically reasoning Network Intrusion Detection Networks allows effective collaboration of individual system defenders that may match a market-based collaboration structures of the attackers.

**Acknowledgment.** This material is based upon work supported by the ITC-A of the US Army under Contract W911NF-12-1-0028 and by ONR Global under Contract N62909-12-1-7019. 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 US Government. Also supported by Czech Ministry of Education grant AMVIS-AnomalyNET: MSMT ME10051 and by Czech Ministry of Interior grant number VG20122014079.

## References


1. Cisco netflow, `http://www.cisco.com/warp/public/732/tech/netflow`
2. Aumann, R.: Correlated equilibrium as an expression of Bayesian rationality. Econometrica: Journal of the Econometric Society (1987)
3. Blum, A., Mansour, Y.: Learning, regret minimization and equilibria. In: Algorithmic Game Theory, ch. 4, pp. 79–101. Cambridge University Press (2007)
4. Debar, H., Curry, D., Feinstein, B.: The intrusion detection message exchange format (idmef). rfc 4765 March, (4765) (2007)
5. Elshoush, H.T., Osman, I.M.: Alert correlation in collaborative intelligent intrusion detection systems–a survey. Applied Soft Computing (2011)
6. Hart, S.: Adaptive Heuristics. Econometrica 73(5), 1401–1430 (2005)
7. Hart, S.: Nash equilibrium and dynamics. Discussion Paper Series dp490, Center for Rationality and Interactive Decision Theory, Hebrew University, Jerusalem (2008)
8. Hart, S., Mas-Colell, A.: A simple adaptive procedure leading to correlated equilibrium. Econometrica 68(5), 1127–1150 (2000)
9. Huynh, T.D., Jennings, N.R., Shadbolt, N.R.: Fire: An integrated trust and reputation model for open multi-agent systems. In: ECAI, pp. 18–22 (2004)
10. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory. Cambridge University Press, New York (2007)
11. Rehak, M., Pechoucek, M., Grill, M., Stiborek, J., Bartos, K., Celeda, P.: Adaptive multiagent system for network traffic monitoring. IEEE Intelligent Systems 24(3), 16–25 (2009)
12. Shamma, J., Arslan, G.: Dynamic fictitious play, dynamic gradient play, and distributed convergence to Nash equilibria. IEEE Transactions on Automatic Control 50(3), 312–327 (2005)
13. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press (March 1998)
14. Wunder, M., Littman, M.L., Babes, M.: Classes of multiagent q-learning dynamics with epsilon-greedy exploration. In: ICML 2010, pp. 1167–1174 (2010)
15. Zinkevich, M., Johanson, M., Bowling, M.H., Piccione, C.: Regret minimization in games with incomplete information. In: Platt, J.C., Koller, D., Singer, Y., Roweis, S.T. (eds.) NIPS. MIT Press (2007)

# Shedding Light on Log Correlation in Network Forensics Analysis

Elias Raftopoulos, Matthias Egli, and Xenofontas Dimitropoulos

ETH Zurich, Switzerland
{rilias,fontas}@tik.ee.ethz.ch, eglima@ee.ethz.ch

**Abstract.** Presently, forensics analyses of security incidents rely largely on manual, ad-hoc, and very time-consuming processes. A security analyst needs to manually correlate evidence from diverse security logs with expertise on suspected malware and background on the configuration of an infrastructure to diagnose if, when, and how an incident happened. To improve our understanding of forensics analysis processes, in this work we analyze the diagnosis of 200 infections detected within a large operational network. Based on the analyzed incidents, we build a decision support tool that shows how to correlate evidence from different sources of security data to expedite manual forensics analysis of compromised systems. Our tool is based on the C4.5 decision tree classifier and shows how to combine four commonly-used data sources, namely IDS alerts, reconnaissance and vulnerability reports, blacklists, and a search engine, to verify different types of malware, like Torpig, SbBot, and FakeAV. Our evaluation confirms that the derived decision tree helps to accurately diagnose infections, while it exhibits comparable performance with a more sophisticated SVM classifier, which however is much less interpretable for non statisticians.

**Keywords:** Network forensics, IDS, Malware, Infections.

## 1 Introduction

Computer Security Incident Response Team (CSIRT) experts use a combination of intuition, knowledge of the underlying infrastructure and protocols, and a wide range of security sensors, to analyze incidents. Although, the low-level sensors used provide a source of fine-grained information, often a single source is not sufficient to reliably decide if an actual security incident did occur. The process of correlating data from multiple sources, in order to assess the security state of a networked system based on low-level logs and events is complex, extremely time consuming and in most parts manual. Although, thorough manual investigation is critical in order to collect all the required evidence for a detected breach and to make a definite assessment regarding the severity of an investigated incident, it would be highly beneficial for administrators to have tools that can guide them in the log-analysis process, helping them to diagnose and mitigate security incidents.

U. Flegel, E. Markatos, and W. Robertson (Eds.): DIMVA 2012, LNCS 7591, pp. 232–241, 2013.

A large number of previous studies have analyzed how to aggregate, correlate, and prioritize IDS alerts. A survey can be found in [24]. However, aggregated IDS alerts are then passed to a security analyst for manual diagnosis, which is a complex, typically ad-hoc process that leverages multiple security sources, like blacklists, scanning logs, etc. In this work we focus on this process with the goal of understanding how to correlate *multiple* security data sources and how to expedite manual investigation.

For this purpose, we conduct a complex experiment turning a human security analyst into the subject of our analysis. We systematically monitor the used evidence and the decisions of an analyst during the diagnosis of 200 security incidents over a period of four weeks in a large academic network.

Based on the analyzed incidents, we build a decision tree using the C4.5 algorithm that reflects how low-level evidence from the four security sources can be combined to diagnose different families of malware, like Torpig, SbBot, and FakeAV. The derived model is useful for expediting the time-consuming manual security assessment of security incidents. It accurately encodes a large part of the decisions of the analyst in correlating diverse security logs and can serve as a decision support tool helping an analyst identify the most critical features that suggest the presence of an infection. In addition, we show that using the decision tree for fully-automated classification correctly identifies infections in 72% of the cases.

Finally, we ask the question if other state-of-the-art classifiers exhibit better performance than a C4.5 decision tree, which is highly interpretable and therefore useful as a decision support tool. We compare its detection accuracy with a support vector machine (SVM), a Bayesian tree classifier (BTC), and a tree-augmented naive Bayes (TAN). We find that a C4.5 decision tree is better than BTCs and TANs and only slightly worse than the more sophisticated SVM, which however is much less interpretable.

In summary, in this work we make the following contributions:

- We outline a number of features useful for security assessment that can be extracted from four commonly-used data sources.
- We build a decision support tool that clearly depicts how evidence from four sources should be correlated to diagnose different types of malware. We show that our decision tree is 72% accurate in automatically classifying suspected infections.
- We compare our decision tree with other classifiers and show that state-of-the-art SVMs, which are more sophisticated but much less understandable, have only slightly better performance.

In the next section we describe in detail the data and features we used. In Section 3 we provide a brief summary of our experiment. Next, in Section 4 we present our decision support tool and in Section 5 we compare its performance to other alternatives. Finally, in Section 6 we discuss related work and we conclude in Section 7.

## 2 Data Sources and Feature Extraction

In this section, we review in detail the data we used and the features we extracted. We conducted all our experiments in the network of the main campus of the Swiss Federal Institute of Technology at Zurich (ETH Zurich). We use four data sources. IDS alerts provide a view of malicious activity from the gateway of the studied network. Reconnaissance and vulnerability reports provide fine-grained information about local hosts, like the client or server role of a host, the running services, and the associated vulnerabilities. Finally, blacklists and search engine queries provide two additional views that are particularly useful for remote hosts. More details about the extracted features can be found in [22].

### 2.1 IDS Alerts

Our IDS data is comprised of raw IDS alerts triggered by a Snort sensor [25] that monitors all the upstream and downstream traffic through the main border link of the network of the main campus of ETH Zurich. The sensor is configured with the official Snort signature ruleset and the Emerging Threats (ET) ruleset [7], which are the two most commonly-used Snort rulesets.

We use IDS alerts in two ways. First, IDS alerts form the input to an IDS alert correlator we developed in [21], which detects infected hosts that exhibit a recurring multi-stage alert pattern involving specific classes of alerts. In particular, the correlator first aggregates similar alerts, then it classifies aggregate alerts into three classes relating to an *Attack*, a *Compromised host*, or a *Policy*, and finally it uses alerts of the first two classes to detect internal hosts that exhibit a recurring multi-stage alert pattern. During our experiment, the alert correlator processed 37 million Snort alerts and detected 200 infected hosts, which were thoroughly analyzed further using data from four security sources.

Secondly, we further exploit IDS alerts during the manual investigation of suspected hosts. Given the IP address of an infected host and the timestamp of the infection, we retrieve the aggregate IDS alerts of the classes *Attack* and *Compromised host* that were observed within 24 hours before or after the timestamp. From the aggregate IDS alerts of these two classes we extract the following features:

- *Suspicious remote hosts*: If the communication to a remote host triggers more than 10% of the total number of aggregate alerts of a local host, we deem the remote host suspicious and mark its IP address for further investigation. We select the 10% threshold empirically based on the alert volume distribution for remote hosts. This feature is useful to identify common malicious domains used by infected hosts to receive instructions, share data, or update their malicious binary.
- *Suspicious remote services*: We aggregate the activity of all non-privileged ports (port numbers above 1024) into a single port with label *High*. If more than 10% of the aggregate alerts target a specific remote port, then we consider this service suspicious. This feature is useful to identify targeted

services, e.g., worms performing a distributed scan for vulnerable services or spamming bots.
- *Suspicious local services*: If a local service port is involved in more than 10% of the aggregate alerts, then we consider it suspicious. Again, we aggregate the activity of all non-privileged ports into a single port with label *High*. This feature is useful to detect malware that attach to popular software such as IE, Firefox, Skype, or CuteFTP.
- *Count of severe alerts*: We count the total number of aggregate alerts. This feature is important to detect malware that generate spurts of high severity alerts. It helps to distinguish high activity malware from more stealthy ones.
- *Infection duration*: We compute the time in hours that elapsed between the first and the last triggered alert within the observed daily interval. We only take into account hourly slots where at least one alert from any class, including policy alerts, was triggered. This feature enables to normalize the volume of alerts of a suspected host over time.
- *Common severe alerts*: If a specific alert accounts for more than 5% of total number of aggregate alerts, then we build a new feature for its alert ID. This feature targets malware that have a very consistent network footprint triggering always the same set of IDS alerts.

### 2.2 Reconnaissance and Vulnerability Reports

We next actively probe suspicious internal local hosts to collect information about running services and vulnerabilities. We use this information to evaluate if the software and operating system (OS) a node is susceptible to the malware reported in the corresponding IDS alerts. We first scan a host using Nmap, and then we use the Nessus [11] and OpenVas [12] vulnerability scanners to build a comprehensive profile of the vulnerability status of a node.

In summary, we extract the following features from reconnaissance and vulnerability reports:

1. *Host Reachability:* This binary feature indicates if a host is reachable. Nodes behind a firewall or a NAT will typically be unreachable.
2. *Host Role:* We exploit hostname keywords, such as `proxy-XX.ethz.ch` and `guest-docking-nat-YY.ethz.ch`, to determine the role of a host. This feature takes the values *client*, *dns-server*, *web-server*, *ftp-server*, or *unknown-server*.
3. *OS and active services:* For each open service in a host we set a corresponding bit in a bitmap of all observed services. Each bit is treated as a separate feature in our classification. In addition, we use Nmap OS fingerprinting and encode the most likely OS match into an additional feature.
4. *Vulnerability data:* We collect vulnerability reports from Nessus and OpenVas and use this information in the manual diagnosis performed in Section 3. However, we exclude vulnerability data from the feature space used in the classification scheme discussed in Section 4, since it drastically increases the

dimensionality of the input data. For example, we have seen that a host running an unpatched version of Windows 7 typically has more than 60 active vulnerabilities.

### 2.3 Blacklists

The third security source we exploit is blacklists. Blacklists are commonly-used to identify IP addresses and domains that have been reported to exhibit malicious activity. We use them to investigate hosts in the *suspicious remote hosts* feature, which is extracted from IDS alerts. We leverage five public blacklist providers [5,13,10,6,3]. The blacklists are partly labeled providing information about the reason a host was enlisted, including the type of malicious activity it was involved in, e.g., bot activity, active attack, and spamming. For each local host, we lookup the corresponding suspicious remote hosts in the blacklists and count the number of hits with a specific label. The count of each label forms an input feature for our classifier.

### 2.4 Search Engine

A lot of useful information about remote hosts resides on the web coming from several diverse sources such as DNS lists, proxy logs, P2P tracker lists, forums, bulletins, banlists, etc. In order to exploit this information, we query the Google search engine using as input string the IP address of an analyzed host and the respective domain name. For each suspected internal host we query for the contacted IP addresses in the *suspicious remote hosts* feature. Then, in an automated fashion we parse the output and extract tags, like *malware*, *spam*, *trojan*, *worm*, *bot*, *adaware*, *irc*, and *banlist*, based on the methodology of [26]. The list of tags we used can be found in [22].

## 3 Forensics Analysis Experiment

In this section we briefly describe our forensics analysis experiment. More information on the validated infections and the diagnosis process along with four example malware cases can be found in [22].

We used the Snort alert correlator we developed in our previous work [21] to detect infected hosts within the monitored infrastructure. During our experiment, we ran our correlator on the latest Snort alerts and we passed on a daily basis newly detected infections to an analyst for manual inspection and validation. Our experiment lasted for approximately four weeks between 01.04.2011 and 28.04.2011 during which we thoroughly investigated 200 consecutive infections. We limited our study to nodes with static IP addresses, which correspond to the majority of the active nodes within the monitored network. Besides, we built automated tools to extract the features discussed in Section 2 and to present them on a dashboard in order to facilitate the investigation process. The analyst would typically have to check the quality of collected signatures, examine

the network footprint generated by the studied host, gather information about the expected behavior of the investigated malware, and most importantly cross-correlate this information. We validated the activity of a specific malware type for 170 out of the 200 infections. We use this set of validated infections to build our decision support tool in Section 4.

## 4 Decision Support Tool

In this section we introduce a decision tree that captures how to correlate the key evidence from the four security sources to identify different types of malware. Our decision tree is useful for expediting the complex and time-consuming manual correlation of multiple data sources for the diagnosis of infected hosts.

We use the C4.5 decision tree induction algorithm, which is a state-of-the-art tree-based classifier [20]. Studies have shown that its performance is better than BTCs and TANs, whereas it is comparable to SVMs [14]. Moreover, it is computationally efficient and an open-source implementation is publicly available. For our purposes, the most important aspect of C4.5 is the interpretability of its results. It is important that a security analyst can understand which feature contributed in every step of the process of a decision, without requiring expert statistical knowledge such as in the case of SVMs. The classification is performed using a tree, where each internal node corresponds to an intermediate decision based on one or more features and the leaf nodes correspond to the final decision.

We use the J48 implementation of the C4.5 algorithm [27]. It takes as input a sample of vectors that correspond to the manually classified hosts. Each vector captures the values of different security features of a host. We use in total 131 security features we described in Section 2. Secondly, C4.5 takes as input the class of each host.

In Figure 1 we show the derived decision tree. The tree accurately depicts the main decisions of the manual process followed in the investigation of security incidents highlighting the most important signs of infection that can drive forensics analysis. Using the decision tree we can easily identify critical signs of malicious behavior. In the following paragraphs, we provide examples of how to combine evidence to detect specific types of malware.

*Zbot*-infected hosts are prominent spammers. We see that they generate a high percentage of high severity alerts that are related to destination port 25. These correspond to spamming attempts for which Snort raises an alert. Moreover, we see that they typically attempt to share stolen confidential data with an HTTP POST request on a malicious domain. This typically triggers the IDS alert 2013976:"*ET TROJAN Zeus POST Request to CnC*". Periodically, the bot attempts to upgrade its binary triggering alerts with ID 2010448:"*ET MALWARE Potential Malware Download, trojan zbot*".

Besides, *SdBot*-infected hosts exhibit frequent communication with their C&C. The bot attempts to identify a valid communication channel from a set of predefined rendez-vous domains in order to update its instruction set and to potentially post valuable client data it has intercepted. These attempts trigger alerts

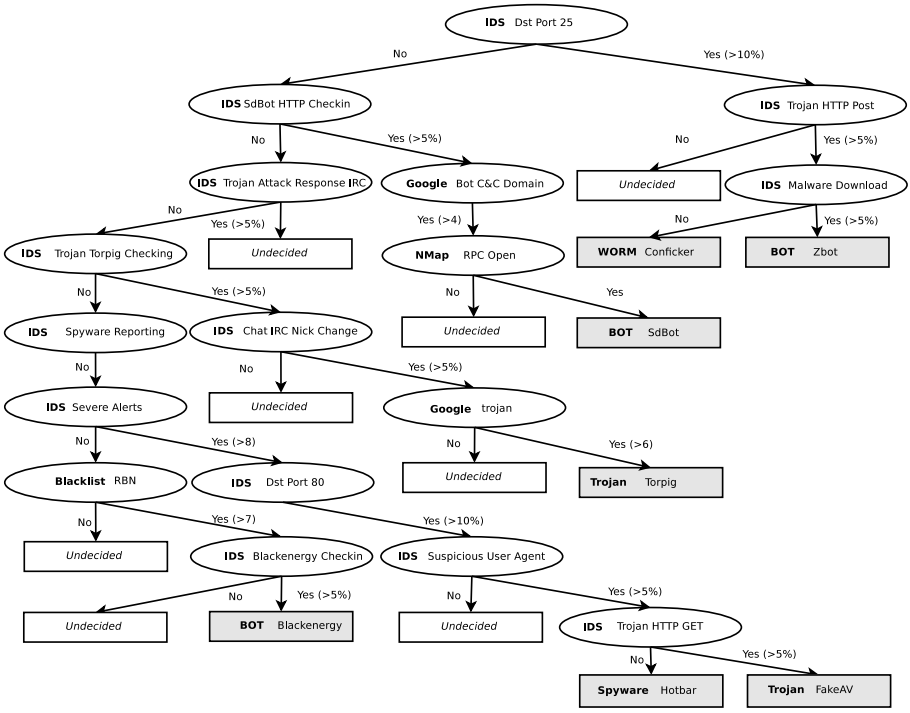



**Fig. 1.** Decision tree generated from the C4.5 algorithm using training data from 200 manually examined incidents

with ID 2007914:"*ET WORM SDBot HTTP Checkin*". Moreover, the malware uses MS network shares to propagate and therefore we see that on most of the infected machines port 135 is open, which corresponds to the RPC service.

The *Torpig* trojan periodically attempts to post using HTTP the data it has stolen from a victim triggering the Snort alert "*ET TROJAN Sinowal/Torpig Checkin*" with ID 2010267. Also, Torpig typically uses IRC to receive updates resulting in frequent IRC nickname changes, detected by the Snort rule 542: "*CHAT IRC nick change*". The domains used to upload harvested user data, i.e., `vgnyarm.com`, `rajjunj.com` and `Ycqgunj.com`, were tagged with the keyword *trojan* by our search results.

Finally, *FakeAV* is a trojan that intentionally misinterprets the security state of the victim and generates pop-ups attempting to redirect the user to domains where the victim can purchase software to remediate the malware. This activity generates a high number of alerts with ID 2002400:"*ET USER_AGENTS Suspicious*" that are related to port 80 (HTTP) activity.

From our analysis for building the decision tree we highlight the following key observations:

– A small number of features is sufficient to make an assessment with high certainty. In most studied cases these features reflect different stages of the

lifecycle of a malware. C4.5 decision trees retain a high level of interpretability assisting the analyst in making an assessment using a manageable number of intuitive features.
– Combinations of features yield more accurate results. Multiple security data sources are required to detect a wider range of malware types exhibiting complex behavioral patterns. In contrast, simple rules of thumb such as 'if the IDS triggers $N$ alerts of type $X$ then there is an infection' cannot be effectively used.

Finally, we note that C4.5 can be used in an adaptive fashion leveraging a feedback loop with the analyst, who can update the set of classified infections in order to enrich the derived tree and improve the classification results. Model induction is very efficient even for datasets involving a large number of features.

## 5 Automated Diagnosis

In this section we analyze the effectiveness of our decision support tool in fully automated diagnosis. We also compare how C4.5 performs against other state-of-the-art classifiers. Specifically, we evaluate the classification accuracy of two tree-based classifiers, namely BTCs and TANs. We use their WEKA implementation with the default parameters. We also evaluate the performance of an SVM, which is a state-of-the-art classifier. To configure the SVM parameters we use the sequential minimal optimization method [17].

**Table 1.** Performance of different classification algorithms.

| Malware Type | C4.5 | | Bayesian Tree | | TAN | | SVM | |
|---|---|---|---|---|---|---|---|---|
| (#incidents) | TP (%) | FP (%) | TP (%) | FP (%) | TP (%) | FP (%) | TP (%) | FP (%) |
| Trojans (85) | 83 | 10 | 80 | 12 | 82 | 10 | 89 | 6 |
| Spyware (59) | 85 | 4 | 85 | 5 | 85 | 4 | 88 | 4 |
| Backdoors (18) | 55 | 8 | 53 | 7 | 56 | 7 | 63 | 5 |
| Worms (8) | 75 | 1 | 75 | 1 | 75 | 1 | 77 | 1 |
| Undecided (30) | 60 | 10 | 48 | 14 | 51 | 13 | 63 | 9 |

In Table 1 we summarize our findings. C4.5 exhibits on average a true positive rate of 72% whereas the false positive rate does not exceed 7%. Bayesian networks and TANs are worse exhibiting a true positive rate of 68% and 70% and a false positive rate of 8% and 7%, respectively. On the other hand the SVM achieves slightly better classification results with a true positive rate of 76% and a false positive rate that does not exceed on average 5%.

## 6 Related Work

Previous studies have extensively studied the aggregation and correlation of IDS alerts with the goal of generating high-level inferences from a large number of low-level alerts. A group of studies exploit statistical correlation to perform causality inference and root cause analysis of detected incidents [23,18,19].

A second group of studies hardcode expert knowledge by introducing scenarios [16,2,15] or sets of rules [1,21] that capture observed malicious behavior. These studies mine solely IDS alerts without taking into account complementary sources of security logs that are often available. They produce and prioritize inferences that at the end are passed on to a security analyst for manual inspection. Our work is complementary and focuses on the manual verification of aggregated IDS alerts by correlating data from multiple instead of a single source.

Besides, a number of commercial solutions, such as IBM Tivoli SCM [9], Alienvault [4], and GFI Languard [8], unify scattered security sensors within an enterprise and provide a single framework that can be used by security analysts to configure security sensors and to visualize logs. However, log correlation in these systems is based on simple rules that need to be determined by an administrator. In our work, we encode a number of classification rules in a C4.5 decision tree that can form the input to such systems.

Finally, in our previous work we developed a Snort alert correlator [21] and we characterized a number of aspects of approximately 9,000 infections we detected over a period of nine months in a large academic infrastructure. We also validated our correlator based on the experiment we describe in this paper. In this work, we describe a number of additional lessons we learned from our experiment and we build a decision support tool to facilitate network forensics analyses.

## 7 Conclusions

Network forensics analysis can be likely better described as art rather than science. It relies on a security expert combining his reasoning with background knowledge about malware, domain specific knowledge about the target environment, and available evidence or hints from a number of diverse security sensors, like IDS systems, vulnerability scanners, and other. We believe that in the future processes for handling and diagnosing security incidents should be largely automated diminishing (but not completely removing) the involvement of humans in the loop.

In this work we analyze the decisions of an analyst during the diagnosis of 200 security incidents over a period of four weeks. We show that a large part of the decisions can be encoded into a decision tree, which can be derived in an automated fashion. The decision tree can help as a decision support tool for future incident handling and provides comparable performance in fully automated classification with a more advanced classifier, an SVM, which however is much less interpretable.

**Acknowledgements.** The authors wish to thank the anonymous reviewers for their helpful comments and suggestions. Furthermore, we wish to thank Prof. Bernhard Plattner and Dr. Vincent Lenders for their invaluable help and fruitful discussions. We would also like to thank Stephan Sheridan and Christian Hallqvist at ETH for their help in the collection and archiving of the data used in this paper.

## References


1. Cuppens, F., Miège, A.: Alert correlation in a cooperative intrusion detection framework. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy (2002)
2. 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)
3. Advanced automated threat analysis system, `http://www.threatexpert.com`
4. AlienVault, `http://www.alienvault.com/`
5. Anonymous postmasters early warning system, `http://www.apews.org`
6. Cooperative Network Security Community, `http://www.dshield.org`
7. Emerging Threats web page, `http://www.emergingthreats.net`
8. GFI Languard, `http://www.gfi.com/network-security-vulnerability-scanner`
9. IBM Tivoli SCM, `http://www-01.ibm.com/software/tivoli/`
10. Shadowserver Foundation web page, `http://www.shadowserver.org`
11. The Nessus vulnerability scanner, `http://www.tenable.com/products/nessus`
12. The Open Vulnerability Assessment System, `http://www.openvas.org`
13. The Urlblacklist web page, `http://www.urlblacklist.org`
14. Friedman, N., Geiger, D., Goldszmidt, M.: Bayesian network classifiers. Mach. Learn. 29, 131–163 (1997)
15. Morin, B., Debar, H.: Correlation of Intrusion Symptoms: An Application of Chronicles. In: Vigna, G., Kruegel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 94–112. Springer, Heidelberg (2003)
16. Ning, P., Cui, Y., Reeves, D.S.: Constructing attack scenarios through correlation of intrusion alerts. In: Proceedings of the 9th CCS ACM Conference (2002)
17. Platt, J.C.: Sequential minimal optimization: A fast algorithm for training support vector machines (1998)
18. Qin, X.: A probabilistic-based framework for infosec alert correlation. PhD thesis, Atlanta, GA, USA (2005) AAI3183248
19. Qin, X., Lee, W.: Statistical Causality Analysis of INFOSEC Alert Data. In: Vigna, G., Kruegel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 73–93. Springer, Heidelberg (2003)
20. Quinlan, J.R.: Induction of decision trees. Mach. Learn. 1, 81–106 (1986)
21. Raftopoulos, E., Dimitropoulos, X.: Detecting, validating and characterizing computer infections in the wild. In: Proceedings of IMC 2011, NY, USA (2011)
22. Raftopoulos, E., Dimitropoulos, X.: Technical report: Shedding light on data correlation during network forensics analysis. TIK Technical Report 346, ETH Zurich (2012)
23. Ren, H., Stakhanova, N., Ghorbani, A.A.: An Online Adaptive Approach to Alert Correlation. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 153–172. Springer, Heidelberg (2010)
24. Sadoddin, R., Ghorbani, A.: Alert correlation survey: framework and techniques. In: Proceedings of PST 2006, pp. 37:1–37:10. ACM, New York (2006)
25. A free lightweight network IDS for UNIX and Windows, `http://www.snort.org`
26. Trestian, I., Ranjan, S., Kuzmanovi, A., Nucci, A.: Unconstrained endpoint profiling (googling the internet). SIGCOMM Comput. Commun. Rev. (2008)
27. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques, 2nd edn. Morgan Kaufmann (2005)

# Author Index