Environment–Reactive Malware Behavior: Detection and Categorization

Smita Naval[1](✉), Vijay Laxmi[1], Manoj S. Gaur[1], Sachin Raja[1], Muttukrishnan Rajarajan[2], and Mauro Conti[3]

[1] Malaviya National Institute of Technology, Jaipur, India
{smita.710,vlgaur,gaurms,sachinraja13}@gmail.com
[2] City University of London, London, UK
r.muttukrishnan@city.ac.uk
[3] University of Padova, Padua, Italy
conti@math.unipd.it

Abstract. Present malicious threats have been consolidated in past few years by incorporating diverse stealthy techniques. Detecting these malwares on the basis of their dynamic behavior has become a potential approach as it suppresses the shortcomings of static approaches raised due to the obfuscated malware binaries. Additionally, existing behavior based malware detection approaches are resilient to zero–day malware attacks. These approaches rely on isolated analysis environment to monitor and capture the run–time malware behavior. Malware bundled with environment–aware payload may degrade detection accuracy of such approaches. These malicious programs detect the presence of execution environment and thus inspite of having their malicious payload they mimic a benign behavior to avoid detection. In this paper, we have presented an approach using system–calls to identify a malware on the basis of their malignant and environment–reactive behavior. The proposed approach offers an automated screening mechanism to segregate malware samples on the basis of aforementioned behaviors. We have built a decision model which is based on multi–layer perceptron learning with back propagation algorithm. Our proposed model decides the candidacy of a sample to be put into one of the four classes (clean, malignant, guest–crashing and infinite–running). Clean behavior denotes benign sample and rest of the behaviors denote the presence of malware sample. The proposed technique has been evaluated with known and unknown instances of real malware and benign programs.

1 Introduction

Widespread use of the Internet and communication technologies has leveraged the malicious attackers to carry out their evil intentions. These malware generated threats disrupt our system services, sensitive information and communication infrastructure. Distribution of these threats makes our system a source of infection spread. To avoid these infections, security researchers across the globe have developed numerous malware detection techniques which rely on signatures

© Springer International Publishing Switzerland 2015
J. Garcia-Alfaro et al. (Eds.): DPM/SETOP/QASA 2014, LNCS 8872, pp. 167–182, 2015.
DOI: 10.1007/978-3-319-17016-9_11

or pattern matching (either static or dynamic). The dynamic behavior–based malware detection (DBMD) approaches have an upper hand as compared to static signature based techniques [1]. The behavior based approaches are capable of capturing the polymorphic, metamorphic, obfuscated and packed variant of known malware. Also, these techniques allow to detect unseen malware attacks [10]. A standard procedure is adopted to monitor the run–time behavior of malware in the most of these DBMD approaches. During the monitoring process, a safe and isolated environment set–up is established to protect the host from any malware generated side–effects. Variety of sandboxing, virtualization and emulator based tools are available to create such set–ups. These controlled environments imitate a real runtime environment. But a full imitation of real system cannot be achieved as these environments differ in CPU semantics (CPU semantic attacks) and in instruction execution time (timing attacks) as compared to uninstrumented host [1]. The detection–aware malware exploits these variations and ensures that it is being analyzed. Rutkowska developed a technique Red Pill [22] which shows that incorporating CPU semantic attack using SIDT (Store Interrupt Descriptor Table) is an effortless task. On the other hand, the timing attacks are deployed using TSC (Time Stamp Counter) register. In virtual machines, the instruction execution time is more than the real machines and this difference can be captured using TSC. Thus, by employing these attacks malware senses the existence of non–real environments and portrays an incorrect picture of itself. Existing plethora of DBMD techniques [10,18,21] do not address this category of malware. These techniques rely on run–time traces and perform the analysis on acquired logs. This form of analysis is not sufficient to alleviate the current environment–aware malware threats. There is a need of an automated approach which can infer the strand of malware infection from their environment–reactive behavior. These observed behaviors can be used to detect and categorize the unknown detection–aware malware instances.

Environment–Reactive Malware Behaviors: Present detection–aware malware carries multiple payloads to remain invisible to any protection system and to persist for longer period of time. The environment–aware payloads determine originality of running environment, if environment is not real then the actual malicious payload is not delivered. After detecting the presence of virtual or emulated environment, malware either terminates its execution or mimics a benign/unusual behavior. In literature, different terms have been utilized to address this category of malware such as environment–sensitive, environment–resistant, environment–aware and split–personality. We have used environment–reactive term for the same. Malware with environment–reactive behavior incorporates two activities (1) sensing the presence of virtual environment and (2) responding to this environment sensing by showing an unusual behavior. We closely monitored these environment–reactive behaviors and labeled each malware binary during training phase. These behaviors are discussed in Sect. 3.

Contributions: In this paper, we have proposed a multi–class model which can identify the environment–reactive behavior of malware binaries. The focus of our proposed approach is to predict the behavior class of a test sample by

neural network based learning algorithm. We have also considered the execution logs of malware and benign executables which do not encapsulate environment–reactive behavior. Using these behavior traits, we have discriminated between clean and malicious applications. Following are the contributions of our proposed approach:

(a) Devised a mechanism which transforms the manual behavior screening into the automated one. Our proposed approach exploits malware's tactics of evading detection to predict its reactive and malicious behavior under a host–based virtualized environment (Ether [7]). Though the proposed approach is specific to a given monitoring environment but can be generalized by applying same methodology with other environments also.
(b) Construction of input vector that is best suited for our learning model. The input vector consists of transition probabilities of two consecutive system–calls.
(c) Creation of an unbiased neural–network based multi–class decision model. We have constructed four networks based on multi–layer perceptron learning algorithm with back propagation. Each network is trained and tested in parallel to reduce the performance overhead.
(d) Our experimental results indicate that the proposed model is capable of (1) finding the known and unknown instances of malware binaries which are environment–reactive (2) improving the monitoring mechanism of Ether by analyzing the detected binaries.

The rest of the paper is organized as follows. Section 2 briefly outlines the related work in analysis–aware malware detection. Section 3 outlines the basic building blocks of our approach. The experimental setup and results are explained in Sect. 4. Section 5 explores the limitations of our approach. Finally, the concluding remarks are presented in Sect. 6.

2 Related Work

The desirable property of any malware detector is that it must be capable of detecting unknown malicious attacks. Modern malware is analysis–aware therefore it contains various other payloads which try to evade the existing detection mechanism. To remain undetected, these malware apply various checks during their execution prior to delivery of the actual payload. Techniques have been illustrated in literature that address the detection of analysis–aware malware.

2.1 Feasibility of Analysis–Aware Malware

Bethencourt *et al.* [3] have demonstrated the feasibility of PIR (Private Information Retrieval)–based malware (specifically worms). The authors discussed that these malicious programs are analysis–resistant and bundled with crpto–computing techniques. They have shown that these malicious codes covertly retrieve the sensitive information from infected computing nodes while hiding their presence from any analysis environments.

2.2 Behavior Divergence Due to Analysis Environment

Chen *et al.* have proposed an approach in [5] to identify the malware behavior under three different environment (virtual, debugger and real). They have compared the behavior of a sample using linear least squares fitting and Mean Squared Error (MSE) to capture malware having anti–debugging and anti–virtualization behavior. Similar work has been proposed in [14,24]. In former approach, the authors have detected the malware which employ anti–VM technique. They have calculated the behavioral distance of a sample using Levesthein distance. The calculated distance depicts the divergence in malware behavior in real and virtual environments. The authors in latter approach have proposed a tool DISARM, to detect the malware with evasive behavior. They have shown that same sample does not constitute similar behavior if executed in different environments. They have calculated a distance score using Jaccard index to explore the change in malware behavior in four different sandboxing environments. Balzarotti *et al.* [2] and Kang *et al.* [11] have analyzed the difference in execution traces of a malware in different analysis. In [2], authors have described the split–personality behavior of malware in virtual and uninstrumented reference host. Any sample having non–similar system–call traces in these two environments are termed as split–personality malware. Kang *et al.* [11] have compared the behavior difference of run–traces of malware in real and emulated environments to detect malware with anti–emulation techniques.

Our proposed approach differs from existing approaches as:

- It does not execute a sample in multiple environments which reduces our execution monitoring time and efforts. We detect the environment–reactive malware behavior from a single execution trace of malware in a single analysis environment.
- It does not incorporate any fingerprint matching or distance matching technique. We developed a multi–class behavior model which detects and categorizes the benign and malicious (Non–environment reactive and environment–reactive) program behavior.

3 Approach Overview

Main objective of our approach is to identify the malware's environment–reactive behaviors. In addition to that, our approach also discriminates the benign and malware (Non environment–reactive) executables. To achieve these objectives, we executed binaries in a virtualized environment using Ether and noted interaction of binary with Kernel in terms of system–calls. On the basis of noted behaviors, we have classified samples in four categories. Figure 1 outlines proposed classification framework details of which are as follows.

3.1 Analysis Framework

The analysis framework plays an important role in analyzing behavior of malicious binaries. It must provide an isolated and transparent environment setup

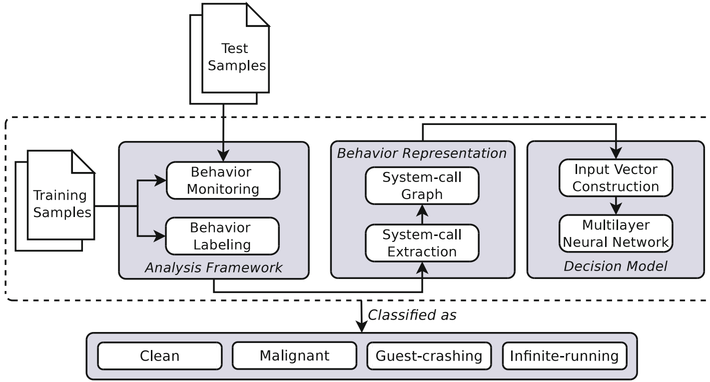


Fig. 1. Proposed classification framework

which does not relay any side–effects to host. We have used Ether as our dynamic analysis framework. It offers a complete and accurate execution sequence by utilizing native CPU instruction and out–of–the guest monitoring [19]. We preferred Ether over other platforms (BitBlaze, Anubis, CWSandbox, Cuckoo) as it has the capability of providing solutions against anti–detection attacks incorporated by malware binaries [1]. These anti–detection attacks include anti-debugging, anti–emulation, anti–sandboxing and host–modification. Our approach relies on Ether for behavior monitoring and labeling of benign and malware executables.

Behavior Monitoring: Ether [7] produces a page fault or exception to intercept the system–call made by the target application. Whenever this application requires a system service, it executes `SYSENTER`. The `SYSENTER` transfers the control from user space to kernel space where it copies the value (address) stored in a special register `SYSENTER_EIP_MSR` into instruction pointer (IP). Ether sets this value of `SYSENTER_EIP_MSR` to a default value. Accessing this value causes a page fault and in this way Ether knows that a system–call has been made. The value `SYSENTER_EIP_MSR` is changed back to its original value and the target application continues its execution. Ether mediates all access to the `SYSENTER_EIP_MSR` register and can therefore hide any modifications of the register from the analysis target [7]. To generate system–call logs, each sample is permitted to execute for 10 min. According to [20], five minute is enough duration for the execution monitoring. We doubled this execution time to capture the malware equipped with capability of carrying out time–out attacks.

Behavior Labeling: According to [7] Ether is capable of capturing execution traces of all malware and remains invisible to the target application being monitored. We have noticed that there are samples for which no logs are generated which indicate that Ether is not completely transparent. Also, Pek *et al.* [19] have shown in their approach (nEther) that Ether is prone to timing attack. In such situations, when Ether is detected by an application the acquired logs cannot be trusted. So, we applied a close manual monitoring of malware behavior of

training samples and utilized these noted behavior to detect the maliciousness of an unknown sample. In training phase, we adapted following definition of clean and malicious behavior of benign and malware binaries.

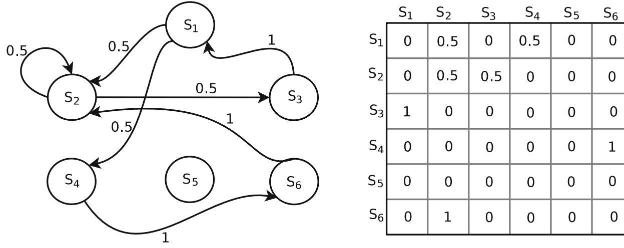
$$\mathbb{P} = \begin{cases} \text{Suspicious} & \{\beta \mid \beta \epsilon \{M,\ G,\ I\}\} \\ \text{Clean} & \text{Otherwise} \end{cases} \quad (1)$$

A program $\mathbb{P}$ is said to be suspicious or clean if it depicts a behavior β in one of the four forms (1) Malignant (M), (2) Guest–crashing (G), (3) Infinite–running (I) and (4) Clean (C). All four are discussed as follows:

- *Malignant (Non Environment–Reactive):* This class of malware depicts the non environment–reactive behavior of malware binary. These malicious programs generate system–call logs within our specified timeframe and do not constitute any visible abnormal activity which alters the guest OS state.
- *Guest–crashing (Environment–Reactive):* Guest–crashing behavior is marked for those samples which crash the guest OS to terminate the execution–monitoring process. To incorporate such a mechanism in the malware source code is not a difficult task. The system can be crashed by causing a page fault, exception and access violation [15].
- *Infinite–running (Environment–Reactive):* We labeled a malware sample as having infinite running behavior if the target sample does not terminate in 10 min and logs show repetitions. Malware binaries eagerly wait just for a single chance of execution and if granted it will try to infect the machine as soon as possible. The infinite running behavior indicates that malware is running in an infinite loop and try to add non–malicious sequence in generated logs.
- *Clean (Non Environment–Reactive):* We marked each benign application with this behavior. The benign executables do not reflect any system crash or infinite running behavior. Our benign dataset does not include any large setup and installation files having execution time more than 10 min.

3.2 Behavior Representation

We have utilized system–calls to model program behavior during execution. System–calls are the interface by which an user application can interact with kernel to access the system services. A program behavior is speculated from the OS state. Any alternation in OS state cannot be made without using system–calls as these are non–bypassable interface [23]. In our approach, the acquired system–call sequences are transformed as system–call graph which is based on Markov model. Markov model based graph representation enables the consolidated comparisons in two dimensional space and maintains the sequential nature of data [1]. Representing system–call sequences in this way hampers malware author's aim of evading detection of any system–call based approach. As the malware authors can very conveniently re–arrange or insert irrelevant system–calls in their malware source code [13]. According to [9] and our traces, there are 284 unique system–calls that can be invoked by any running application in Windows XP.



	S1	S2	S3	S4	S5	S6
S1	0	0.5	0	0.5	0	0
S2	0	0.5	0.5	0	0	0
S3	1	0	0	0	0	0
S4	0	0	0	0	0	1
S5	0	0	0	0	0	0
S6	0	1	0	0	0	0

Fig. 2. System–call graph and TPM: an example

System–Call Graph: Let $\boldsymbol{\xi}$ is an execution trace of a sample which represents the set of system–calls invoked by the sample. $\boldsymbol{\xi}$ is further transformed into weighted directed graph $G = \{V, E\}$, where V is a finite set of 284 unique system–calls and E represents set of edges in G. Every edge e_{ij} indicates a transition from node i to j with transition probability ρ_{ij}. Applying Markov property (Eq. 2), we built our Transition Probability Matrix (TPM).

$$\sum_{j=1}^{284} \rho_{ij} = \begin{Bmatrix} 0 \; if \; all \; entries \; in \; i^{th} \; row \; is \; zero \\ 1 \qquad\qquad otherwise \end{Bmatrix} \tag{2}$$

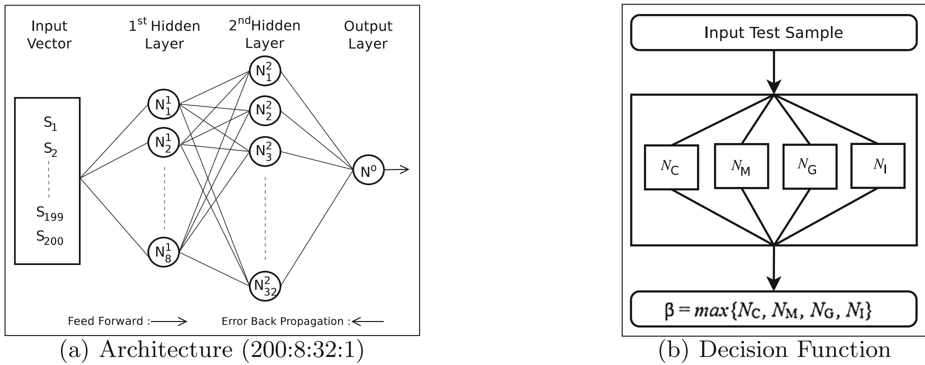
For example, consider the execution trace $\boldsymbol{\xi} = \{S_1, S_2, S_3, S_1, S_4, S_6, S_2, S_2, S_3, S_1\}$ of a program $\mathbb{P}$. Figure 2 shows the corresponding graph and matrix for this example. Here, program $\mathbb{P}$ invokes 5 unique system–calls $(S_1, S_2, S_3, S_4, S_6)$ and call S_5 is not utilized in its execution path. The graph contains 5 connected nodes and one isolated node. Edges are directed (showing transition direction) and labeled with transition probability ρ_{ij}. The matrix representation of $\mathbb{P}$ shows a 6×6 square matrix called as TPM. Every row in TPM is summed to either 1 or 0. In our experimentation, we have constructed 284×284 TPM for each benign and malware executable. These individual TPMs are used to form a composite matrix of a dataset as shown in Eq. 3. Here, composite matrix (R.H.S.) is constructed by adding two TPMs (L.H.S.) and then dividing each cell (i, j) of resultant matrix by number of samples (2 in this case). In similar manner we have created composite matrix of our datasets which is further used to construct our input vector. Each cell in this matrix can have a value ranging from 0 to 1 which indicates the average transition from one state to other in a dataset.

$$\begin{Bmatrix} a_{11} \; a_{12} \; a_{13} \\ a_{21} \; a_{22} \; a_{23} \\ a_{31} \; a_{32} \; a_{33} \end{Bmatrix} + \begin{Bmatrix} b_{11} \; b_{12} \; b_{13} \\ b_{21} \; b_{22} \; b_{23} \\ b_{31} \; b_{32} \; b_{33} \end{Bmatrix} = \frac{1}{2} \begin{Bmatrix} (a_{11} + b_{11}) \; (a_{12} + b_{12}) \; (a_{13} + b_{13}) \\ (a_{21} + b_{21}) \; (a_{22} + b_{22}) \; (a_{23} + b_{23}) \\ (a_{31} + b_{31}) \; (a_{32} + b_{32}) \; (a_{33} + b_{33}) \end{Bmatrix} \tag{3}$$

3.3 Decision Model

We used neural network [8,17] model to categorize the aforementioned behaviors. This model has ability of learning non–linear discriminant function and recognizing patterns in high–dimensional feature space. It can be structured using

either single–layer or Multi–Layer Perceptron (MLP). Our initial experiments with single–layer perceptron indicate that our data is not linearly separable. For this reason, we adopted MLP [8] model with error back propagation algorithm for our classification methodology. We applied one–against–all (OAA) [17] pattern modeling over one–against–one (OAO) and P–against–Q (PAQ) because other two modeling methods require more number of network structures and shall result into a computational overhead. Applying OAA, our network becomes a bi–class model and therefore we developed four different networks for each of the behaviors (C vs. All, M vs. All, G vs. All and I vs. All).



(a) Architecture (200:8:32:1) (b) Decision Function

Fig. 3. Decision model

Figure 3(a) shows our decision model (200:8:32:1). The architecture of our proposed network is described by one input vector, two hidden layers and one output layer. We constructed input vector using state transition probabilities discussed later in Sect. 4.2. The proposed classification model was designed with two hidden layers instead of one according to the findings of the work in [6]. We adopted hit and trial strategy to select the number of neurons in the hidden layer as low and high neuron count may result into under–fitting and over–fitting which further lead towards poor learning of training sets. In this quest, we performed an extensive experimentation by using neuron count from 4 to 40 at both the hidden layers. We observed best results with 200:8:32:1 in terms of accuracy and training time and hence decided for 8 neurons in the first hidden layer and 32 neurons in the second hidden layer. At the output layer, a single neuron is sufficient in our case, since for each network, we need a single value which gives a measure of how closely an input sample relates with the corresponding behavior.

Besides these structural issues there are factors which play a crucial role in designing a decision neural network. These factors are described as follows.

Activation Function: Every neuron is associated with a bias value and makes use of an activation function to transform the value of the activation level into an output signal. If the learning algorithm is back propagation, it is necessary

for the activation function to be differentiable. For our model, we have selected $\frac{tanh(x)}{No.\ of\ neurons\ in\ the\ layer}$ as the activation function as (i) it is highly differentiable and leads to good gradient descend on error, (ii) it requires less mathematical computations on each neuron, and (iii) it gives output in the range -1 to $+1$. We have trained our neural network in a way that a positive value is expected for a positive class and negative value for negative class.

Learning Rate and Momentum Rate: Learning rate is a numerical factor by which weights are updated in each iteration. Lower the learning rate, finer tuned are the weights, but it requires more iterations, thereby leading to a high training time. Higher learning rate results in less overall training time with lesser accuracy of detection. Also, during the learning process a momentum factor is introduced to reduce the sensitivity of the network to a local minima with respect to weights and increases the convergence speed. For our model, we have observed good classification results and appreciable training time with 0.01 as the value of the learning rate and 0.5 as the momentum factor value.

Decision Function: When a sample is to be classified, its output from each network is generated and passed through a decision function for a final classification. A high positive output from a network indicates a close match of the input sample to the corresponding behavior. Figure 3(b) illustrates the process of determining class of a input test sample. The sample is supplied into all four networks and four output values N_C, N_M, N_G, and N_I are generated in parallel with respect to each network. It is labeled with behavior β where β is the maximum value out of four generated values.

4 Experimental Setup and Results

The experiments were performed on Intel Core i7 2.30 GHz with 8 GB, 1600 MHz DDR3 RAM Macbook Pro. Implementation code is written in JAVA (eclipse IDE) and executed in JRE environment with 2.5 GB and 3.0 GB of heap space.

4.1 Dataset Preparation

We have used malware and benign executables as input. Total 1150 Benign executables are gathered from `Windows/system32` directory of freshly installed Windows system. Our proposed model relies on the system–call sequence gathered from a target binary while it is being executed in Windows XP platform. Although Microsoft abandons its support to Windows XP yet this will not affect our proposed model because (i) the target malicious binaries (win32 PE) affect the all Windows platform, (ii) system–call sequence used in Windows XP is a subset of those used in Windows 7 [9].

Our malware dataset consists of 1120 samples collected from on line sources and user agencies. All malicious samples are then applied to three different AV scanners (Norton, Quick Heal, AVG) to segregate them into their respective malware families (Worm, Trojan, Virus). The training (70 %) and known test

sets (30 %) are formed from both the benign and malware datasets. The known test set consists of samples whose behavior is known but these are not used for training. We labeled each known malware and benign sample with their respective behavior. We labeled 1150 samples as Clean, 505 samples as Malignant, 329 samples as Guest–crashing and 286 samples as Infinite–running. It is clear that more than 50 % of malware samples depict environment–reactive behavior reinforcing our motive of detecting environment–reactive malware. Furthermore, we have used one unknown test set, samples of which is not labeled with any behavior to check whether our model makes an accurate behavior prediction for unknown instances.

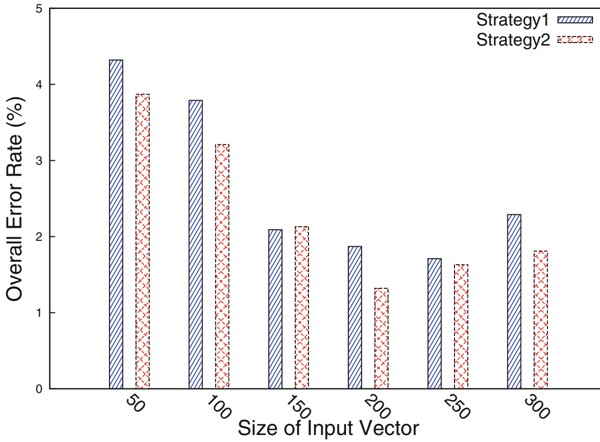


Fig. 4. Input vector construction

4.2 Input Vector Construction

Input vector plays a key role in designing a decision network which yields into a better detection accuracy. We have chosen transition states of composite matrix to be used as input in our constructed input vector. Composite matrix is a 284×284 matrix so our state space will have total 284×284 transitions. This state space is very large and applying it to our model will degrade its performance. Therefore, to decide the appropriate transition states and size of the input vector we adopted following two strategies.

1. Construct *four* input vectors w.r.t. each behavior network using four composite matrices corresponding to each of the behavior datasets. (Strategy 1)
2. Construct *one* input vector for all four behavior networks using one composite matrices created from training samples of all behavior datasets. (Strategy 2)

We selected top 50, 100, 150, 200, 250 and 300 transition states for both the strategies. After this, we applied TPMs of our samples and trained all four networks for fixed 10000 iterations with these strategies. The main aim of this

experiment is to select the suitable strategy and size for our input vector. Figure 4 shows results of this experiment in terms of overall error rate with respect to each of the 12 (two strategies and six sizes) experiments. We found that Strategy 2 outperforms the Strategy 1 as the individual choice of input vector trains the model for relevant behavior class and ignores the global knowledge. Though, there is a marginal difference in the error rate but for any malware detection system this difference cannot be avoided. Therefore, we considered Strategy 2 for our experimentation. The minimum error rate is obtained at input size 200 for Strategy 2. We observed that the error rate increases as we increase the input size. Because adding more state to our input vector will also increase the noise in the data and as a result it will lead to poor detection accuracy. Also, we observed that the training time is directly proportional to input size. But, we decided to sacrifice training time over detection accuracy and fixed the input vector size as 200.

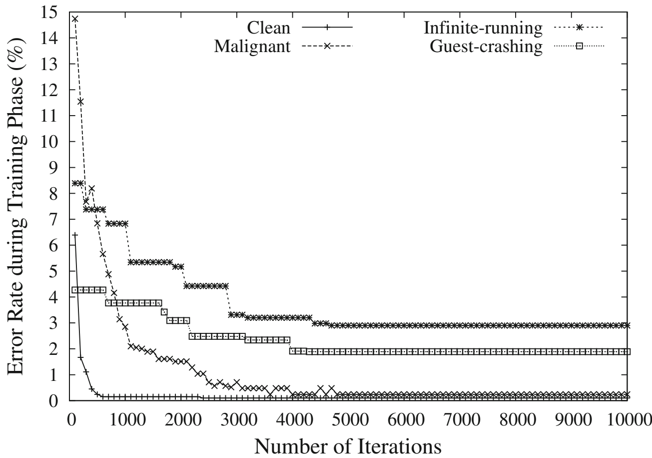


Fig. 5. Number of iteration selection

4.3 Training Results

In training phase, we randomized samples after each epoch to avoid the biasness in the adjusted weights. Figure 5 shows the plot of error rate vs. number of iterations. This figure indicates that our model has been trained in just 5000 iterations as the overall error rate get stabilized from this point onwards. It will allow us to fix the number of iterations to 5000 for our testing phase. As discussed earlier, the training time increases as we increase the number of iterations therefore selecting low number of iteration for our model will improve the running performance of the model. But, this selection cannot assure an accurate decision model as the error stability is must in such type of networks. We can ignore the training time as it is one time cost to achieve higher detection accuracy for unknown test samples.

We evaluated our proposed model's performance using TPR, FPR, TNR, FNR, Accuracy and Error [25]. Table 1 illustrates performance of constructed trained neural networks. A high value of TPR and low value of FPR indicates that developed model is performing well. Our model discriminates the clean and malignant behavior with higher accuracy of 99.9 % and 99.24 %. The false–alarm rate in these cases is negligible which indicates that our selected input vector is significantly diverse in identifying samples with clean and malignant behavior. Remaining two behavior classes indicate the categorization within a malware class, so here we expected the overlapping. The infinite–running and guest–crashing are two environment–reactive behaviors which do not reflect its malicious behavior instead these categories of malware try to mimic the clean behavior. Therefore, we are observing a false–alarm rate in these two cases.

Table 1. Performance evaluation with training and known test datasets

Dataset	Network	TPR	FPR	TNR	FNR	Accuracy
Training	C vs All	99.8	0	100	0.2	99.9
	M vs All	98.56	0.09	99.91	1.44	99.24
	I vs All	95.12	0.92	99.08	4.88	97.1
	G vs All	97.01	0.24	99.76	2.99	98.39
Test	C vs All	97.22	3.4	96.6	2.78	96.91
	M vs All	97.12	1.59	98.41	2.88	97.76
	I vs All	95.56	3.04	96.96	4.44	96.26
	G vs All	92.8	5.53	94.47	7.2	93.64

4.4 Testing Phase

We conducted testing for our known and unknown test datasets. As illustrated earlier, the known test samples are labeled with their respective behavior but are not utilized during the training phase. Table 1 shows the results of our known test dataset. We observed that the testing results are quite similar to the training results. There is a tolerable difference in detection accuracy of our training and test datasets because of our model is trained with less number of guest–crashing and infinite–running samples as compared to benign samples. We have observed that overall testing accuracy of 96.125 %.

Table 2 shows our testing results with unknown test set. We supplied this set to verify the correctness of proposed model with unknown unlabeled instances of binaries. From statistics, we can deduce that our model is capable in categorizing the unknown instances. We confirmed the assigned behavior labels of unknown test samples by monitoring these samples in Ether. With an error rate of 4.6 %, we found that designed model automates our manual behavior labeling process. This proves that the proposed model can determine the environment–reactive behavior of unknown instances also.

Table 2. Detection accuracy with unknown test dataset

Behavior class	# Samples	# Correct instances	# Incorrect instances
Clean	265	258	7
Malignant	103	102	1
Guest–crashing	36	34	2
Infinite–running	19	17	2

4.5 Performance Evaluation

To evaluate the performance of the proposed supervised learning model we considered three factors. These factors are described as follows.

Detection Rate: The detection rate determines the accuracy of a proposed model. The proposed model addresses the malware detection problem which categorizes the malware instances according to their suspicious (Non–environment–reactive and environment–reactive) behavior. This detection problem is sensitive therefore the high true positive rate and low false positive rate is desirable. We obtained high TPR and low FPR in Table 1.

Generality: Generality determines how well the trained model performs on test data. According to [17] we cannot achieve the generality with smaller (as in our case) and imbalanced datasets. However, our experimental results contradict this and indicate significant uniformity in testing and training results. For instance the FNR value with both training and test samples tables are more as compared to FPR. The reasons for this achieved generality are that (i) our constructed input vector includes such patterns which can identify the diversity in all four behavior classes and (ii) our dataset is not completely imbalanced. To check whether the dataset is balanced or not we applied an imbalance measure derived in [17]. Let Ω be an imbalance measure for OAA. Equation 4 decides the value of Ω. Here $K = 4$ denotes the number of classes and n_i is the total number of samples in i^{th} class. If Ω tends to zero means the dataset is imbalanced and the maximum value for Ω will be $1/(K-1)$ which denotes that the dataset is balanced. In our case, Ω is 0.15 which indicates that our training set is not completely imbalanced and thus we have achieved uniformity in training and testing results.

$$\Omega = \boldsymbol{min}\left\{ \frac{n_i}{\sum_{j=1,\ i\neq j}^{k} n_j} \,\middle|\, i = 1, ..., K \right\} \quad (4)$$

Training Time: We created neural networks equal to the number of behaviors to be detected, which in our case is 4. In order to achieve a high efficiency with respect to training time, we trained all these four networks in parallel by making use of JAVA threads. Hence, training of each network occurs in parallel and the overall training time becomes equal to the maximum value of the time taken by all four networks. We obtained training time in the range of 17–1096 s for 100–10000 iterations respectively with heap space 3 GB. We also trained the designed

model with heap space 2.5 GB (training time ranges from 22 to 1536 s) and observed a significant speed up (1.45 for 5000 iterations) with the former. This speedup is achieved due to the lower execution frequency of the JAVA garbage collector in higher heap space. The obtained training time is not a big issue to be considered. But when our proposed method applied with larger datasets this time will increase drastically. To reduce the training time with larger dataset we can increase the heap space memory of the system and can acquire a significant speed up without making any modifications into our developed model.

5 Limitations

We have shown that our approach can detect real unknown malware on the basis of their environment–reactive behavior. Also we have observed that most of the current malware depicts these behaviors. However, malware portraying behavior other than malignant, guest–crashing and infinite cannot be detected by our decision model. To overcome this, the proposed model can be trained again to detect the new behavior as well. In this section, we describe cases which restrict our detection and categorization approach.

Tracing and behavior screening of malware binaries completely depend on Ether. So, the limitations associated with Ether are inherited into our approach. The malware with root privileges can detect Ether framework as it utilizes the emulated version of Bochs virtual machine [12] for BIOS data strings. Our proposed model does not deal with these obscure malware as Ether cannot generate their execution logs. Second, analyzing executables from Ether is a time consuming task. Ether uses exceptions whenever a running application makes a system–call to access system service. These exceptions result into significant performance overhead.

Our approach does not consider multiple execution paths of running binaries. Exploring multiple execution path of a single sample can detect the trigger–based malware. The trigger–based malware delivers its malicious payload only if certain trigger conditions are met. These triggers include certain system state, timestamp (a particular date, day or even time), URL and certain keywords [4,16]. Our single–execution path based approach may miss this category of malware. However, our approach can detect a fraction of these trigger–based malware which prefer environment–sensitive payload over trigger–based payload in its execution path. Also, our experimental results deduce that a single execution path of an executable is sufficient to detect its benign or malignant nature.

6 Conclusion

Emerging malicious threats are detection–aware therefore the current DBMD approaches are in getting out–of–date. New generation of malware can detect the presence of analysis–environment and do not reflect malignant behavior. Ether which is a hardware–based virtualization framework can also be detected by these new–generation malware samples. In our proposed approach, we developed

a decision model for identifying and categorizing the behavior of a sample during its execution in virtualized environment. The proposed model automates our behavior screening and labeling process and captures the environment–reactive and non–environment–reactive behavior of malware. It makes use of execution sequence generated in a single analysis environment and this log sequence is used to predict the behavior class of malware. We evaluated our proposed model with known and unknown real instances and discovered that the proposed model is effective in detecting these instances.

Acknowledgments. Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission under the agreement No. PCIG11-GA-2012-321980. This work is also partially supported by the TENACE PRIN Project 20103P34XC funded by the Italian MIUR, and by the Project "Tackling Mobile Malware with Innovative Machine Learning Techniques" funded by the University of Padua.

References

1. Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic analysis. J. Comput. Virol. **7**(4), 247–258 (2011)
2. Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., Vigna, G.: Efficient detection of split personalities in malware. In: Proceedings of the Network and Distributed System Security Symposium, NDSS, San Diego, California, USA, pp. 1–16 (2010)
3. Bethencourt, J., Song, D., Waters, B.: Analysis-resistant malware. In: Proceedings of the Network and Distributed System Security Symposium, NDSS, San Diego, California, USA, pp. 1–13 (2008)
4. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Lee, W., Wang, C., Dagon, D. (eds.) Botnet Detection. Advances in Information Security, vol. 36, pp. 65–88. Springer, New York (2008)
5. Chen, X., Andersen, J., Mao, Z., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Dependable Systems and Networks With FTCS and DCC, DSN, pp. 177–186, June 2008
6. Chester, D.L.: Why two hidden layers are better than one. In: Proceedings of the International Joint Conference on Neural Networks, IJCNN 1990, Washington, DC (1990)
7. 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. ACM, New York (2008)
8. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**(5), 359–366 (1989)
9. J00ru: Windows win32k.sys system call table, April 2014
10. Jacob, G., Hund, R., Kruegel, C., Holz, T.: Jackstraws: picking command and control connections from bot traffic. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, pp. 29–48. USENIX Association, Berkeley (2011)
11. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating emulation-resistant malware. In: Proceedings of the 1st ACM Workshop on Virtual Machine Security, VMSec 2009. ACM, New York, pp. 11–22 (2009)

12. Kevin, L., Bryce, D., David, G., Volker, R., Christophe, B.: Bochs user manual (2010)
13. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009, pp. 351–366. USENIX Association (2009)
14. Lindorfer, M., Kolbitsch, C., Milani Comparetti, P.: Detecting environment-sensitive malware. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 338–357. Springer, Heidelberg (2011)
15. Mark, R., David A, s., Alex, L.: Windows internal part 2
16. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP 2007, Washington, DC, pp. 231–245 (2007)
17. Ou, G., Murphey, Y.L.: Multi-class pattern classification using neural networks. Pattern Recogn. **40**(1), 4–18 (2007)
18. Park, Y., Reeves, D.S., Stamp, M.: Deriving common malware behavior through graph clustering. Comput. Secur. **39**, 419–430 (2013)
19. Pék, G., Bencsáth, B., Buttyán, L.: nEther: in-guest detection of out-of-the-guest malware analyzers. In: Proceedings of the Fourth European Workshop on System Security, EUROSEC 2011, pp. 3:1–3:6. ACM, New York (2011)
20. Quist, D., Liebrock, L., Neil, J.: Improving antivirus accuracy with hypervisor assisted analysis. J. Comput. Virol. **7**(2), 121–131 (2011)
21. 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)
22. Rutkowska, J.: Red pill... or how to detect vmm using (almost) one cpu instruction
23. Srivastava, A., Lanzi, A., Giffin, J.T.: System call API obfuscation (extended abstract). In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 421–422. Springer, Heidelberg (2008)
24. Sun, M.K., Lin, M.J., Chang, M., Laih, C.S., Lin, H.T.: Malware virtualization-resistant behavior detection. In: Proceedings of the 17th International Conference on Parallel and Distributed Systems (IEEE), ICPADS 2011, Washington, DC, USA, pp. 912–917 (2011)
25. Vinod, P., Laxmi, V., Gaur, M.S.: REFORM: relevant feature for malware analysis. In: Proceedings of Sixth IEEE International Conference of Security and Multimodality in Pervasive Environment (SMPE 2012), pp. 26–29. Fukuoka Institute of technology (FIT), Fukuoka, Japan (2012)