applied sciences

MDPI

Article

HM^3alD: Polymorphic Malware Detection Using Program Behavior-Aware Hidden Markov Model

Asghar Tajoddin and Saeed Jalili *

Faculty of Electrical and Computer Engineering, Tarbiat Modares University, P.O. Box 14115-194 Tehran, Iran; a.tajoddin@modares.ac.ir

* Correspondence: sjalili@modares.ac.ir; Tel.: +98-21-8288-4935

Received: 24 May 2018; Accepted: 25 June 2018; Published: 26 June 28

check for updates

Abstract: Malware have been tremendously growing in recent years. Most malware use obfuscation techniques for evasion and hiding purposes, but they preserve the functionality and malicious behavior of original code. Although most research work has been mainly focused on program static analysis, some recent contributions have used program behavior analysis to detect malware at run-time. Extracting the behavior of polymorphic malware is one of the major issues that affects the detection result. In this paper, we propose HM^3alD, a novel program behavior-aware hidden Markov model for polymorphic malware detection. The main idea is to use an effective clustering scheme to partition the program behavior of malware instances and then apply a novel hidden Markov model (called program behavior-aware HMM) on each cluster to train the corresponding behavior. Low-level program behavior, OS-level system call sequence, is mapped to high-level action sequence and used as transition triggers across states in program behavior-aware HMM topology. Experimental results show that HM^3alD outperforms all current dynamic and static malware detection methods, especially in term of FAR, while using a large dataset of 6349 malware.

Keywords: polymorphic malware detection; program behavior-aware hidden Markov model; dynamic analysis; system call monitoring; action sequence

1. Introduction

Endpoint security is regarded as the most important and the last defense point in security threats [1]. According to this requirement, malware detection is the vital issue in computer security. Today, malware are assumed as essential threats in the software industry [2]. In an annual report, Symantec mentions that just in 2015 alone more than 430×10^6 malware variants were created [3]. In this regard, many methods have been proposed that focus on detecting and classifying malware [1]. Due to the increasing growth of malware, anti-viruses are usually unable to completely detect them, because malware programs usually attempt to hide themselves using obfuscation methods so they are hard to detect by static analysis [4].

1.1. Malware

The terms malicious software and malware refer to any computer program that performs malicious activities on a host or accesses a private computer system to gather sensitive information from users without their knowledge. From the beginning of malware's existence, they used code obfuscation to evade detection. Using the obfuscation technique, malware authors generate new malware variants of known malware and easily bypass detection methods. Polymorphism is another attribute that is commonly employed by malware. Polymorphism is an encryption technique that is used to mutate the static binary code of malware to prevent their detection [5]. When an infected program is run, the malware is decrypted and loaded into memory and then infects other programs and/or any type of executable content and tries to run a new version of itself [5]. These malware use a permutation engine to produce a new encryption procedure at runtime. If a malware presents a new behavior (by new functionalities or by combining the features of existing malware), it is called a zero day malware. Malware variants refer to all new malware that are produced manually or automatically from any existing malware [4,5].

1.2. Malware Analysis and Detection

Malware analysis is the process of understanding malware behavior and how to detect and eliminate them by capturing the important characteristics of a given malware sample [6]. This is particularly important for preventing and detecting future cyber attacks against the host and network. There are two main methods of analyzing malware, known as static and dynamic analyses. Static analysis (i.e., code analysis) examines the sample program without running it and inspects the program's binary code to determine its behavior. Static analysis explores all possible execution paths in a program—not just those invoked during execution—but it cannot deal with malware employing anti-reverse engineering technologies such as code packing and obfuscation [6]. Dynamic analysis (i.e., behavior analysis) executes the program in a controlled environment and monitors its behavior. Thus, dynamic analysis explores what function with what arguments is called and detects most obfuscation attempts [6]. The combination of these two methods, hybrid method, could certainly further improve the detection results [7].

Based on malware analysis type, malware detection methods in general fall into two categories: signature-based and anomaly-based. Signature-based (i.e., knowledge or misuse) detection methods use some common sequence patterns (i.e., signatures found in the binary code of malware instances) to identify malware. Most often, signature-based detection methods are very fast because they do not run samples to identify malware. Note that this method is used on most popular commercial antivirus software. The main drawback of signature-based methods is that they are not effective against polymorphism and obfuscation methods, so they cannot detect modified and unknown malicious executables [8,9].

Anomaly-based detection methods build a reference model for the normal behavior of benign programs and look for deviation of programs from the normal behavior to detect them as malware [10]. While anomaly-based methods can detect unknown and zero-day malware, their main weakness is their high false positive rate [10,11].

In this paper, we propose HM^3alD, a dynamic malware detection method based on HMM and high level actions to detect polymorphic malware such as bots, worms, viruses and Trojan horses. Specifically, the proposed method includes the following steps in the training phase: (1) It extracts high level action sequences from system call sequences correspondingly. (2) It clusters only malware action sequences (i.e., malware programs) to group similar sequences. (3) It uses HMM as a one-class classifier to learn the model of malware action sequences per each cluster. (4) It calculates the decision threshold of each malware HMM to discriminate between benign and malware programs. Finally, the detection phase of HM^3alD gives the action sequence of a ran program to all learned HMMs and receives the probabilities returned from them. HM^3alD detects the ran program as benign, if all of the probabilities are less than the corresponding decision thresholds of learned HMMs, otherwise it is considered as malware.

HM^3alD outperforms important previous dynamic and static malware detection methods especially in term of FAR that is a hard work to decrease it without sacrificing DR, and its advantages are: (1) high detection rate; (2) low false alarm rate; (3) low performance overhead; and (4) near to online malware detection.

1.3. Contributions

In the proposed method, HMM is used to stochastically represent program behavior using traces of action sequences issued by processes at run time. The main contributions of this paper are: (1) HMM topology is devised based on program stages (i.e., initialization, running, and termination) to achieve low false alarm rate and high detection rate; and (2) HMM is applied on malware action sequences (derived from system call sequences of programs) to detect polymorphic malware dynamically and decrease the complexity of training phase dramatically. As a result, the proposed method is scalable because it uses only 26 actions (i.e., observations in HMM) that are not increased by increasing the number of malware samples.

1.4. Paper Structure

The rest of paper is organized as follows: Section 2 presents the research literature of malware analysis and detection methods. Section 3 briefly describes the necessary background. In Section 4, the proposed method is explained along with an analysis of its time complexity. In Section 5, we evaluate the performance of HM^3alD using a large dataset and compare it with other methods. In Section 6, we analyze and discuss the idea behind HM^3alD. In Section 7 the paper is concluded.

2. Related Work

We introduce research that detects polymorphic and metamorphic malware by behavioral methods. According to the type of analysis, we divide this section into two parts: "static analysis" and "dynamic analysis".

2.1. Static Analysis-Based Methods

Faruki et al. [12] used API call gram (i.e., the sequence of API calls of a program) to detect malware. They first extracted a call graph from the disassembled instructions of a binary program, and then converted the graph to a call gram. Finally, their pattern-matching engine performs the detection, according to the call gram. Unfortunately, the authors did not report their performance overhead. Kalbhor et al. [13] introduced a method to detect metamorphic malware based on HMMs. They analyzed metamorphic malware that are produced by malware generators such as NGVCK. One HMM is trained for each malware generator and finally malware are detected by similarity degree. Wong and Stamp [14] also presented a method to detect metamorphic malware using HMMs. Similar to the previously mentioned method, an offline static analysis based on Opcodes has been used in this work too. Austin et al. [15] proposed a model using HMM in which they try to distinguish the compiled codes and the assembly codes written by malware developers. Song and Touili [16] presented a detection scheme based on model checking. In this work, a program and its behavior are explained using formal language. Then, the program behavior is checked to detect malware. Shahid et al. [5] proposed a detection framework that uses control graphs to detect metamorphic malware online. Ding et al. [17] proposed QOOA that is an API-based association mining method for malware detection. Hellal et al. [18] presented a new graph mining method to detect variants of malware using static analysis. They proposed a novel algorithm, called minimal contrast frequent subgraph miner algorithm (MCFSM), for extracting minimal discriminative and widely employed malicious behavioral patterns which can identify an entire family of malicious programs.

2.2. Dynamic Analysis-Based Methods

Park et al. [19] proposed a method that clusters polymorphic worms. It uses system call graphs. Thus, it has a high complexity, but its false alarm rate is zero. Shahzad et al. [20] used process control blocks (PCB) at runtime and recorded runtime information of processes such as memory data and execution addresses. They detected malware by decision trees. Elhadi et al. [21] proposed a method to build call graphs of programs at runtime and distinguish malware and benign programs by comparing their call graphs with known malware call graphs. Shehata et al. [22] presented a detection method based on observing system calls at runtime. They mapped system calls to some high-level actions and considered them as features to learn decision trees in order to detect malware. Salehi et al. [23] proposed a dynamic malware feature selection method, called MAAR, based on the name of API calls and their arguments and/or return values recorded during runtime. Several well-known classifiers such as Random forest (RF), Decision Trees, Sequential minimal optimization (SMO), and Bayesian logistic regression (BLR) are used in this study. Christodorescu et al. [24] designed an algorithm to build a call graph by which its nodes are system calls and its edges correspond to the interdependency of system calls. Kolbitsch et al. [25] also generated behavioral graphs by using taint analysis without considering system call arguments. In these graphs, only the data dependencies are considered. Ding et al. [26] built a common behavior graph for each malware family. They used a dynamic taint analysis technique to find the dependency relations between system calls, and then built a system call dependency graph by tracing the propagation of the taint data. Based on the dependency graphs of malware samples, they proposed an algorithm to extract the common behavior graph to detect malware.

3. Background

In this section, we introduce some basic concepts and techniques used throughout this paper.

3.1. Malware Obfuscation

Obfuscation is a technique that obscure the control flow and data structures of a program without differing in its functionality and behavior. Originally, this technology was introduced for the intellectual property of software developers, but it has been broadly used by malware authors to evade from detection engines. Obfuscation techniques are classified into the three categories: "data obfuscation", "static code rewriting", and "dynamic code rewriting". Data obfuscation modifies the form in a program storing the data to hide it from direct analysis. Static code rewriting is similar to compiler optimization, as it modifies program code during obfuscation without any further modifications at runtime. Dynamic code rewriting modifies programs such that the executed code differs from the code that is statically visible in the executable [27].

Polymorphic malware is a type of malware that constantly changes its identifiable features with the help of the obfuscation methods to elude detection. Even though the polymorphic malware effectively thwarts the signature based detection techniques relied on by security solutions such as antivirus software [28].

3.2. Hierarchical Clustering

Clustering is defined as an unsupervised learning to find groups such that objects in the same group (called a cluster) are more similar to each other than to those in other groups. Among many approaches for clustering, hierarchical clustering only uses similarities of objects, without any other requirement on the data [29].

There are two hierarchical clustering algorithms: divisive and agglomerative. A divisive clustering algorithm follows the top-down approach, starting with a single group and breaking up large groups into smaller groups, until each group contains a single object or it meets certain termination conditions. The agglomerative clustering algorithm follows the bottom-up approach, starting with groups, each initially containing one training object, and then merging similar groups into larger groups, until there is a single one or certain termination conditions are satisfied. At each iteration of an agglomerative clustering algorithm, the two closest groups are selected to merge based on similarity measures or links. In single-link clustering, the distance between groups is defined as the smallest distance between their closest objects of the two groups. In complete-link clustering, the distance between groups is taken as the farthest distance between their objects of the two groups [29].

Choosing the number of clusters in a dataset is a fundamental issue. There are various ways to fine-tune the number of clusters. One of the common methods is the "elbow method". In this method, first the reconstruction error or log likelihood is plotted as a function of k (i.e., the number of clusters) and then the "elbow" points are sought as an indicator of the appropriate number of clusters. In this method, the number of clusters is chosen such that adding another cluster does not give much better modeling of the data [29].

3.3. Hidden Markov Model

Markov models [29] are state machines in which the current state depends on previous states statistically. In a first order Markov model, the next state depends on only the current state. A hidden Markov model (HMM) is a statistical Markov model in which the states are not observable directly so they are called hidden states. In summary and formally, HMM includes the following elements, the set of N distinct states: $S = \{s_1, s_2, \ldots, s_N\}$; the set of M distinct observations in each state: $V = \{v_1, v_2, \ldots, v_M\}$; A: state transition probabilities, $A = [a_{rj}]$ where $a_{rj} = P(q_t + 1 = S_j | q_t = S_r)$; B: observation probabilities, $B = [b_j(m)]$ where $b_j(m) = P(O_t = v_m | q_t = S_j)$; π: initial state probabilities; and $\pi = [\pi_r]$ where $\pi_r = P(q_1 = S_r)$. Recall that q_t denotes the state at time t where $t = 1, 2, ..., N$. Thus, an HMM, $\lambda = (A, B, \pi)$, is defined by A, B, and π (and implicitly by M and N dimensions). For a set of observation sequences $\mathcal{X} = \{O^l\}_l$, an HMM λ is trained such that $P(\mathcal{X}|\lambda)$, the probability $\mathcal{X}$ generated from λ is maximized. Then, for any given observation sequence O^l, and the learned model λ, HMM by applying a forward algorithm finds the corresponding state sequence $Q = \{q_1, q_2, \ldots, q_T\}$ of O^l, such that it maximizes $P(O^l|\lambda)$.

The last issue is HMM topology, which is defined by the number of the states and their connections. Three kinds of general topologies can be found: "Bakis topology", "left–right topology" and "fully connected topology" [30]. In Bakis topology, the rule is: $a_{rj} > 0$ only for $j = r$ or $j = r + 1$. In the left–right topology, the rule is: $a_{rj} > 0$ for $j \geq r$; and in the fully connected topology the rule is: $a_{rj} > 0$ for any r, j. The third topology is also called the ergodic model. Note that the HMM topology can be serial or parallel. In a parallel mode, a sequence of states is parallelized with another sequence of states.

4. Proposed Method: HM^3alD

HM^3alD dynamically detects polymorphic malware based on system calls. It comprises two main phases: the training phase and detection phase. Figure 1 shows the architecture of the HM^3alD method. In the following subsections, we describe each of these phases in detail.

4.1. Training Phase

In the training phase, first we collect the behavior (i.e., sequence of system calls) of malware programs in a controlled environment called sandbox and we extract high-level action sequences from system call sequences. After that, we cluster all malware action sequences. Therefore, we get some malware clusters where the action sequences of each cluster are very similar to each other. We denote the malware cluster set by $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$ where k is the number of malware clusters. Then, we consider a fraction of action sequences of each malware cluster c_i as training observation sequences, $\mathcal{X}_{c_i} = \{O^l\}_{l=1}^{|\mathcal{X}_{c_i}|}$ and train its corresponding HMM, λ_{c_i}, where $O^l = \{O^l_1, O^l_2, \ldots, O^l_{T_l}\}$ is the lth action sequence and $|\mathcal{X}_{c_i}|$ (cardinality of $\mathcal{X}_{c_i}$) denotes the number of training action sequences in cluster c_i. Finally, we compute a decision threshold, $\mathcal{T}_{c_i}$, for each malware HMM, λ_{c_i}, based on another fraction of c_i, called V_{c_i} and some benign action sequences, called V_b.

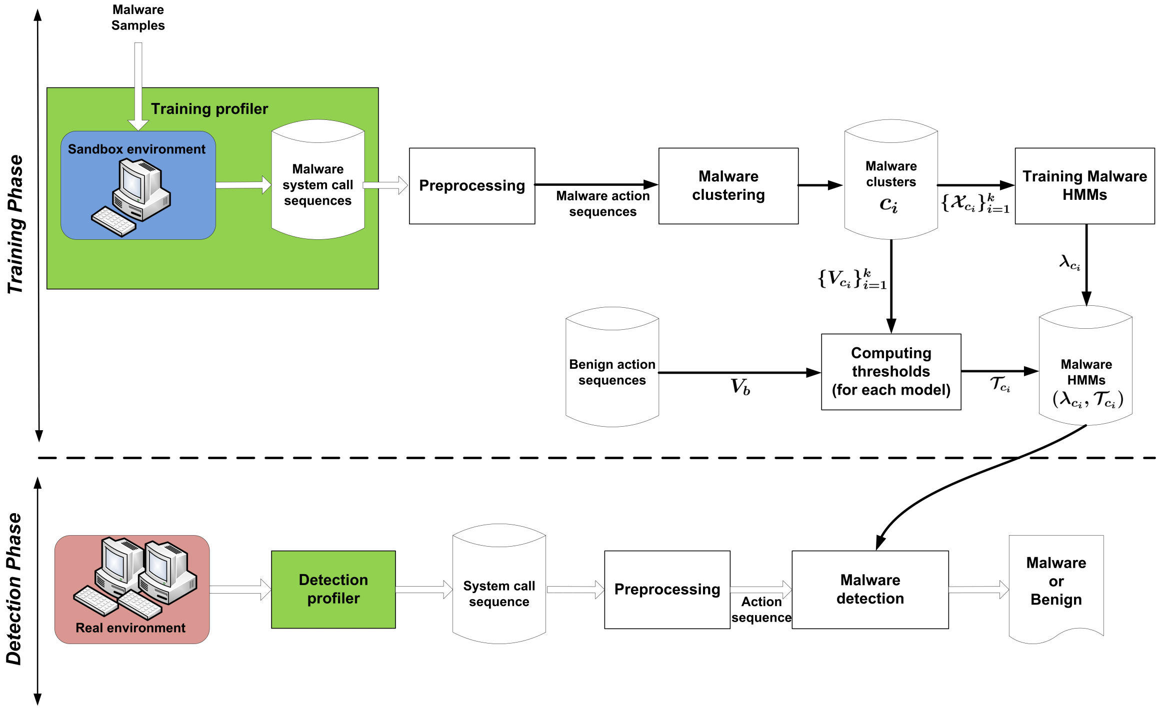


Figure 1. The architecture of HM^3alD.

4.1.1. Training Profiler

In dynamic program analysis, we monitor programs' executions and extract their system calls using "API (Application programming interface) hooking" [31]. This technique allows us to intercept all system call sequences made by running processes. A system call is basically a user request to the kernel of the operating system to get some services, such as opening and closing files, creating and executing a process, or accessing network resources. Sandbox [32] is used to record system calls of a program that are being executed. The training profiler module provides an application level virtualization using sandbox.

4.1.2. Preprocessing

System call sequence is an important resource for dynamic malware detection, but it is too fine-grained, so we map a system call sequence $S = \{s_1, s_2, \ldots, s_n\}$ to a high-level action sequence $AS = \{v_1, v_2, \ldots, v_T\}$ where each action v_i is a subsequence of system calls. Twenty-six actions are defined that are classified in seven sets and are described in Appendix A. The process of generating an action sequence from a system call sequence is presented by Algorithm 1.

4.1.3. Clustering Action Sequences

Since learning techniques (i.e., HMM) have high performance when input samples are more similar to each other and since malware programs may have different classes of action sequences, we therefore partition all action sequences of malware set to some clusters (i.e., $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$) using complete-link agglomerative hierarchical clustering method [33], where each cluster has high cohesion (i.e., high similarity) and less similarity with other clusters. In the process of clustering, we compute the normalized similarity of every two action sequences AS_i and AS_j based on Equation (1). Here, $ED(AS_i, AS_j)$ is the edit distance of AS_i and AS_j, which is computed by the Levenshtein technique [34].

$$Sim(AS_i, AS_j) = 1 - \frac{ED(AS_i, AS_j)}{max(length(AS_i), length(AS_j))}. \tag{1}$$

Algorithm 1 : Preprocessing, generating an action sequence from a system call sequence

Input:

S = A system call sequence $S = \{s_1, s_2, \ldots, s_n\}$.

Output:

AS = An action sequence for system call sequence S.

Define an empty array $\mathcal{B}$ such that each array element specifies a system call sequence during the execution of algorithm.
Initialize A with 26 base actions: $A = \{v_1, v_2, \ldots, v_{26}\}$
for each $s_i \in S$ **do**
  ℓ = **OS_Handle**(s_i) ▷ get the Operating System handle pointer of s_i
  insert($\mathcal{B}(\ell)$,s_i) ▷ for any system call, insert it to $\mathcal{B}(\ell)$
  if s_i is a releasing system call **then** ▷ A releasing system call releases the dependent resources and invalidate the handle.
    v = **Match**(A,$\mathcal{B}(\ell)$) ▷ Match $\mathcal{B}(\ell)$ to the corresponding action in A by a hash function
    insert(AS,v) ▷ Insert action (v) to AS
  end if
end for

In the hierarchical clustering method, we plot the reconstruction error as a function of k and look for an elbow. Then, we set k at elbow. k_Set denotes the set of all candidate k values that points to the one of the elbow points. In HM^3alD, the principle rule is to select the smallest possible k that leads to set of suitable HMMs. We select the smallest k such that: (1) $k \in k\_Set$; and (2) there is only at most one cluster c_i where its members are not similar enough and has just few action sequences. Note that, according to our experience on HMM, if the average similarity of a cluster is less than 0.5, then the corresponding HMM is not converged accurately and leads to a high false alarm rate.

4.1.4. Training HMMs

This section is presented in three parts: (1) input sequences; (2) HMM topology; and (3) yraining.

Input sequences: Action sequences of programs at runtime are observation sequences.

HMM topology: We define the topology of HMMs based on program behavior. Recall that each program consists of three main stages at runtime: (1) initialization; (2) running; and (3) termination. In the initialization stage, each program sets its variables by initial values and allocates its required resources. Thus, we consider this stage as following the serial Bakis topology. In the running stage, programs perform some iterative units of work in the form of conditions and loops. Thus, we consider this stage as following the parallel fully connected (ergodic) topology. The termination stage is somewhat similar to the initialization stage: it publishes the program outputs, deallocates the resources, and terminates the program. Thus, we consider this stage as following the Bakis topology. Figure 2 expresses the novel program behavior-aware HMM topology where the number of states at running stage should be determined.

Training: To learn an HMM, $\lambda_{c_i} = (\pi_{c_i}, A_{c_i}, B_{c_i})$, on each malware cluster c_i, we must set its initial state probability $\pi_{c_i} = \{\pi_1 = 1\}$; its transition probability matrix $A_{c_i} = \{a_{rj} = P(q_{t+1} = S_j | q_t = S_r)\}$; and its observation probability matrix $B_{c_i} = \{b_j(v_m) = P(O_t = v_m | q_t = S_j)\}$. To learn HMM λ_{c_i}, we first determine the number of states N_{c_i} and then initialize A_{c_i} and B_{c_i} matrices.

According to the proposed program behavior-aware topology (Figure 2), matrix A_{c_i} consists of some transitions so the weights of other transitions are zero. Thus, we initialize the values of its non-zero elements randomly with constraint $\sum_{j=1}^{N_{c_i}} a_{rj} = 1$. We initialize matrix B_{c_i} randomly with constraint $\sum_{m=1}^{M} b_j(v_m) = 1$, where $M = 26$ (i.e., the number of actions). Figure 3 shows a six-state HMM topology and its corresponding matrix A.

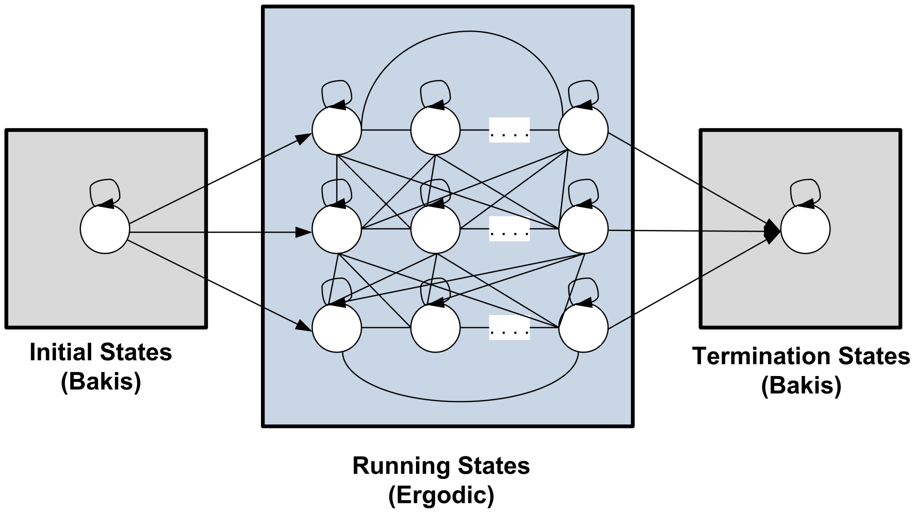
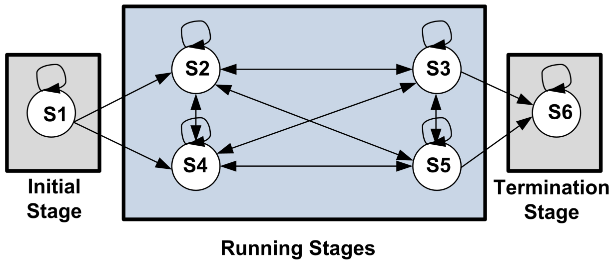


Figure 2. The proposed program behavior-aware HMM topology in HM^3alD.



Matrix *A*

	S1	S2	S3	S4	S5	S6
S1	0.24	0.32	0	0.44	0	0
S2	0	0.23	0.26	0.16	0.35	0
S3	0	0.11	0.17	0.27	0.24	0.21
S4	0	0.37	0.12	0.24	0.27	0
S5	0	0.18	0.24	0.35	0.13	0.10
S6	0	0	0	0	0	1

Figure 3. A six-state program behavior-aware HMM topology and its corresponding transition probability matrix A.

To estimate the number of states of each cluster (i.e., N_{c_i}), we compute the number of unique actions that are observed across all sequences corresponding to that cluster. Since they are based on three-stage program behavior-aware topology, the minimum and maximum values of N_{c_i} are 3 and 26, respectively.

After calculation of N_{c_i} and initialization of A_{c_i} and B_{c_i}, we apply the Baum–Welch algorithm [29] on each cluster c_i to train (the parameters of) its corresponding HMM, λ_{c_i}. The Baum–Welch algorithm is an EM-like algorithm and guarantees to converge towards local optima. The Baum–Welch algorithm iterates the E step and the M step, yielding monotonically increasing log-likelihoods, and the algorithm is terminated when the difference of two subsequent log-likelihoods falls below ε that is near to zero or the maximum number of iterations is met. Typically, the algorithm reaches different local maxima or saddle points for different initializations, so we run it multiple times on each cluster c_i with different initializations and we finally select HMM, λ_{c_i} that has the highest probability. Algorithm 2 shows the steps of training HMMs.

4.1.5. Computing Decision Thresholds

Recall that, in HM^3alD, we learn only malware HMMs λ_{c_i}, to distinguish between malware and benign programs at detection phase. In this section, we compute a threshold $\mathcal{T}_{c_i}$ on each HMM λ_{c_i} using another fraction of malware action sequences and some benign action sequences. To present well the process of computing the decision thresholds, first we devise some definitions about the HMM threshold concept.

Algorithm 2 : HM^3alD-Training phase

Input:

$\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$ denotes the set of malware clusters

Output:

Trained HMMs set, $TrainedHMMs = \{\lambda_{c_1}, \lambda_{c_2}, \ldots, \lambda_{c_k}\}$.

```
TrainedHMMs = ∅                                              ▷ an empty set
for i = 1 to |C| do
    Estimate the number of states (N_{c_i}) for cluster c_i
    Initialize π_{c_i}, A_i and B_i using the proposed program behavior-aware topology
    Compute λ_{c_i}(A_i, B_i) using Baum–Welch algorithm
    Add λ_{c_i} to TrainedHMMs
end for
```

Definition 1. *For each malware HMM λ_{c_i} and its corresponding threshold $\mathcal{T}_{c_i}$, the pair $(\lambda_{c_i}, \mathcal{T}_{c_i})$ is a discriminator such that an observation (action) sequence O^l is identified as a malware action sequence if Equation (2) is satisfied. Note that, from now on, we use $LP(O|\lambda)$ instead of $log(P(O|\lambda))$ for simplicity.*

$$LP(O^l|\lambda_{c_i}) \geq \mathcal{T}_{c_i} = \begin{cases} 1, & \text{if } O^l \text{ is a malware action sequence} \\ 0, & \text{otherwise, no decision} \end{cases}. \tag{2}$$

In Equation (2), if the output is zero, the result (benign/malware) depends on the other HMM decisions.

Definition 2. *For each malware HMM λ_{c_i}, its corresponding rejection rate R_{c_i} is a certain fraction of V_{c_i} (the validation part of observation sequences of malware cluster c_i) that specifies the maximum permissible error on HMM λ_{c_i}. For simplicity, we set $R_{c_1} = R_{c_2} = \ldots = R_{c_k}$, which leads to different values of the decision thresholds $\mathcal{T}_{c_i}$.*

Definition 3. *For each malware HMM λ_{c_i}, the maximum benign probability Pb_{c_i} is the greatest log probability that is returned by HMM λ_{c_i} for V_b, benign action sequences, and is computed by Equation (3).*

$$Pb_{c_i} = \max_l \{LP(O^l|\lambda_{c_i}), \text{ for all } O^l \in V_b\}. \tag{3}$$

The pseudo-code of computing decision thresholds is shown in Algorithm 3.

To compute threshold $\mathcal{T}_{c_i}$ by using Algorithm 3, we proceed as follows. First, we estimate the log probabilities of the all action sequences belong to V_{c_i} and reorder them in ascending order (Lines 3–8). Then, we obtain the malware threshold point tm (Lines 9–10). In many conditions, we find that the Pb_{c_i} is less than or equal to tm, which means that the corresponding HMM λ_{c_i} is sufficiently powerful and returns small log probabilities for benign action sequences. Thus, we finally compute $\mathcal{T}_{c_i}$ based on this condition (lines 12–16).

Algorithm 3 : Computing decision thresholds for all malware HMMs, λ_{c_i}

Input:

$fracrej = R_{c_1} = R_{c_2} = \ldots = R_{c_k}$.
$V_C = \{V_{c_1}, V_{c_2}, \ldots, V_{c_k}\}$ and each $V_{c_i} = \{O^l\}_{l=1}^{|V_{c_i}|}$ is the malware validation part of cluster c_i.
$V_b = \{O^l\}_{l=1}^{|V_b|}$, is a set of benign action sequences.
$Malware\_HMM = \{\lambda_{c_1}, \lambda_{c_2}, \ldots, \lambda_{c_k}\}$, learned HMM set corresponding to malware clusters.

Output:

$TV = \{\mathcal{T}_{c_1}, \mathcal{T}_{c_2}, \ldots, \mathcal{T}_{c_k}\}$, the threshold vector corresponding to malware HMMs. In other words, $\{(\lambda_{c_1}, \mathcal{T}_{c_1}), (\lambda_{c_2}, \mathcal{T}_{c_2}), \ldots, (\lambda_{c_k}, \mathcal{T}_{c_k})\}$

for $i = 1$ to k **do**
  $TV = \emptyset$ ▷ an empty set
  $PV = \emptyset$ ▷ an empty set
  for each $O^l \in V_{c_i}$ **do**
    $Pm = LP(O^l|\lambda_{c_i})$
    insert(PV,Pm)
  end for
  sort(PV) ▷ sort in ascending order
  t_index=**round**($fracrej \times |PV|$)
  $tm = PV(t\_index)$
  Compute Pb_{c_i} by using Equation (3)
  if $Pb_{c_i} \leq tm$ **then**
    $\mathcal{T}_{c_i} = Pb_{c_i}$
  else
    $\mathcal{T}_{c_i} = tm$
  end if
  insert(TV,$\mathcal{T}_{c_i}$)
end for

4.2. Detection Phase

In the detection phase, as shown in the bottom part of Figure 1, for each program P_l at runtime, first its corresponding system call sequence is collected (by *detection profiler* module), and then its corresponding action sequence O^l is generated (by *preprocessing* module). The induced action sequence O^l is given to all learned malware HMMs, $\{(\lambda_{c_1}, \mathcal{T}_{c_1}), (\lambda_{c_2}, \mathcal{T}_{c_2}), \ldots, (\lambda_{c_k}, \mathcal{T}_{c_k})\}$ and then HM^3alD aggregates their returned results by Algorithm 4. Note that, in Algorithm 4, the forward values are calculated by multiplying small probabilities, and with long action sequences we risk getting underflow. To avoid this, at each time step in the forward algorithm, we normalize the forward values. This technique is presented as Algorithm 5 in Appendix B.

Algorithm 4 applies HMM forward algorithm, **forward**(O^l,λ_{c_i}) on the induced action sequence O^l from the running of program P_l and it calculates $LP(O^l|\lambda_{c_i})$ iteratively. The result will be *"Malware"* when at least one of the malware HMMs return the probability value $forward\_value_i$ greater than or equal to the corresponding threshold value $\mathcal{T}_{c_i}$, otherwise it will be *"Benign"*.

4.3. Time Complexity Analysis

In this subsection, we analyze the time complexity of HM^3alD. Recall that HM^3alD consists of two main phases: training and detection. In the following, we discuss the time complexity of these steps in detail.

4.3.1. Training Phase

Without loss of generality, we focus on the core of the training phase: Clustering malware action sequences and Training HMMs. For an initial training set X of N_t samples generated from the malware program dataset, first we partition X to the k malware cluster. The complexity of the naive complete-link agglomorative algorithm becomes $\mathcal{O}(N_t^3)$. Because we exhaustively scan the $N_t \times N_t$ matrix for the largest similarity in each of $N_t - 1$ iterations [29]. After clustering malware action sequences, we train HMMs for each of them using Baum–Welch algorithm. if the number of HMM states is N and the length of an action sequence is T, then the time complexity of Baum–Welch algorithm is $\mathcal{O}(N^2T)$ [29]. We suppose the average length of each malware action sequence is T and, without loss of generality, the average number of states of all malware clusters is N. Thus, the time complexity of Algorithm 2 becomes $\mathcal{O}(N_tN^2T)$. Therefore, we conclude that the overall time complexity of the training phase is $\mathcal{O}(N_tN^2T + N_t^3)$.

4.3.2. Detection Phase

In the detection phase, Algorithm 4 applies HMM forward algorithm to the action sequence O^l and then aggregates their returns. The time complexity of the forward algorithm is $\mathcal{O}(N^2T)$ [29]. Therefore, the time complexity of the detection phase for action sequence O^l becomes $\mathcal{O}(kN^2T_l)$, where N, T_l, and k are the average number of states of all malware clusters, the length of action sequence O^l, and the number of malware clusters, respectively.

Algorithm 4 : *HM^3alD*-Detection phase

Input:

O^l = an induced action sequence O^l from the running of program P_l in the real environment.
$Malware\_HMM = \{(\lambda_{c_1}, \mathcal{T}_{c_1}), (\lambda_{c_2}, \mathcal{T}_{c_2}), \ldots, (\lambda_{c_k}, \mathcal{T}_{c_k})\}$, learned HMM set corresponding to malware clusters.

Output:

program type: Malware/Benign

/*Compute $LP(O^l|\lambda_{c_i})$ of action sequence O^l with the forward algorithm, for all $i = 1 \ldots k$ malware HMMs.*/

```
for i = 1 to k do
    forward_value_i = forward(O^l, λ_{c_i})
/* The forward() is presented as Algorithm 5 in Appendix B.*/
    if forward_value_i ≥ T_{c_i} then
        return "Malware" & exit          ▷ returns "Malware" and then exit.
    end if
end for
/* If none of the malware HMMs detect the program P_l as malware, then the algorithm decides
that it is a benign program. */
return "Benign"
```

5. Experimental Evaluation

This section is composed of: (1) introducing dataset; (2) experimental setup; (3) evaluation metrics; (4) presenting the performance of HM^3alD with different settings; and (5) the comparison of HM^3alD with the state-of-the-art methods.

5.1. Dataset

We used a dataset that consists of 9025 programs, such that 6349 of them are polymorphic malware including bots, worms, viruses and Trojan horses, and the rest (2676) are benign programs. The malware programs are downloaded from VX Heaven virus collection [35] and belong to different families. Each family includes polymorphic samples of a malware that malware writers have made gradually and then they have been registered in VX Heaven's dataset. For the benign program set, several applications were downloaded from sourceforge.net [36]. These applications also fall into different categories including: video and audio, scientific, educational, games, communications, etc.

5.2. Experimental Setup

Detection in HM^3alD was performed at runtime, so recording the system call sequence of a running program must be started when the program begins to run. HM^3alD uses Cuckoo sandbox tool which provides an isolated and safe environment to run programs. All the benign and malware programs are executed under Cuckoo sandbox tool [32] on a host, in a virtual environment. A machine with Intel Core i7-4790K processor and 16 GB RAM was used to execute all experiments. We installed the sandbox tool under Ubuntu-13.03 on this machine. The guest OS in this experiment was Windows XP 32 bit. For each program, Cuckoo sandbox restores the guest OS to a safe state, and then, after executing the program, it returns the corresponding system call sequence in JSON format comprehensively. To extract the action set, we developed a set of basic tools in C++ and they are executed in the sandbox environment. We implemented Algorithm 1 as a python script to drive the action sequence of a program from its corresponding system call sequence. We clustered benign (malware) action sequence sets using MATLAB toolbox. We used GHMM library with python wrapper [37] to implement HMM algorithms.

To evaluate the performance of HM^3alD, we used cross-validation strategy. The malware program set was randomly partitioned into two parts, 70% for learning malware HMMs λ_{c_i} and computing the corresponding malware decision thresholds, and 30% for detection phase. Similarly, the benign program set was randomly partitioned into two parts, i.e. 30% for computing the decision thresholds and 70% for detection phase. To train good HMMs, λ_{c_i}, we randomly repeated the cross-validation strategy 10 times. Note that, in the learning process, Algorithm 2 (Baum–Welch algorithm) was repeated 30 times.

5.3. Evaluation Metrics

According to the general definitions, true positive (TP) is the number of truly detected malware programs, true negative (TN) is the number of truly recognized benign programs, false positive (FP) is the number of benign programs that are detected as malware, and false negative (FN) is the number of malware that are detected as benign programs. N_m and N_b are the number of malware and benign programs, respectively.

Accuracy (Equation (4)) indicates the ratio of malware and benign programs that are truly identified. TP rate (Equation (5)), also called detection rate (DR), is the proportion of malware that are recognized as malware and FP rate (Equation (6)), also called false alarm rate (FAR), is the proportion of benign programs that are incorrectly detected as malware.

$$Accuracy = \frac{TP + TN}{N_m + N_b}, \tag{4}$$

$$DR = \frac{TP}{N_m}, \tag{5}$$

$$FAR = \frac{FP}{N_b}. \tag{6}$$

Note that one-class classification methods typically suffer from a relatively large value of FAR, because they learn from just samples with the same label [38]. Thus, we intended to maximize DR and Accuracy and minimize FAR.

5.4. The Performance of HM^3alD

In this section, the different parameters of the proposed method are determined and the results of HM^3alD, in different conditions, are presented.

5.4.1. Training Phase

First, we determined the basic parameters such as the number of malware clusters and the number of states for each malware HMM. Then, we clustered malware action sequence sets, and, according to the program based HMM topology (Figure 2), we trained HMMs λ_{c_i} on the training part of all malware clusters $\{\mathcal{X}_{c_i}\}_{i=1}^{k}$. Finally, we applied Algorithm 3 to compute their corresponding decision thresholds, $\mathcal{T}_{c_i}$.

Determining the number of malware clusters

According to the cross-validation strategy, we should choose different values for k in hierarchical agglomerative clustering method. As stated in Section 4.1.3, we applied the complete-link agglomerative clustering method on malware action sequence set. Then, for different values of k, its results in terms of reconstruction error are presented in Figure 4.

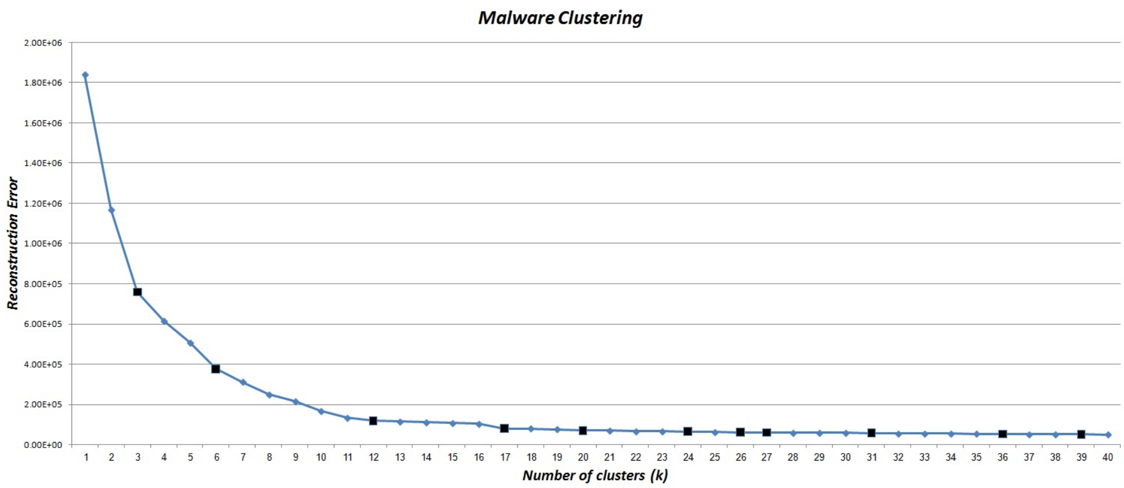


Figure 4. Variation of reconstruction error for different number of malware clusters. Black points shows candidate k values.

As Figure 4 shows, different values can be considered for k. According to the elbow method (see Section 4.1.3), the suitable value of k is one of the candidate values of k_Set = {3, 6, 12, 17, 20, 24, 26, 27, 31, 36, 39}. As mentioned in Section 4.1.3, we tended to select the smallest k from the k_Set such that it satisfies Conditions (1) and (2). To summarize, Tables 1 and 2 show the results of two values of k in which $k = 17$ satisfies the corresponding conditions but $k = 12$ does not. In other words, by considering $k = 12$, we have a cluster (i.e., Cluster 4) such that its size is great and its centroid is less than 0.5 (highlighted row in Table 1). Thus, we can choose one of the k values that is greater than or equal to 17. According to the experiments, which are discussed in Section 5.4.2, the best value of k is 24.

Training HMMs

As in Section 4.1.4, first we calculated the number of states of each cluster (i.e., N_{c_i}). Table 3 presents the number of states for each malware cluster as their corresponding unique actions on training data. According to Figure 2 and Table 3, we design program behavior-aware topology of each HMM, λ_{c_i}. Then, by applying Algorithm 3, we computed their corresponding decision thresholds, $\mathcal{T}_{c_i}$ and finally built all malware HMMs, $(\lambda_{c_i}, \mathcal{T}_{c_i})$. Note that we only use 15% of benign action sequences to compute decision thresholds, $\mathcal{T}_{c_i}$.

Table 1. The malware clusters for $k = 12$.

Cluster No.	Cluster Size	Centroid
1	273	0.65
2	1638	0.59
3	243	0.89
4	318	0.41
5	311	0.82
6	268	0.81
7	376	0.56
8	510	0.62
9	372	0.73
10	588	0.68
11	592	0.83
12	860	0.95

Table 2. The malware clusters for $k = 17$.

Cluster No.	Cluster Size	Centroid
1	468	0.79
2	1170	0.61
3	54	0.13
4	264	0.58
5	492	0.90
6	100	0.63
7	92	0.72
8	284	0.59
9	308	0.79
10	64	0.56
11	273	0.65
12	243	0.89
13	311	0.82
14	268	0.81
15	510	0.62
16	588	0.68
17	860	0.95

Table 3. The number of HMM states for $k = 24$.

Cluster No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
N_{c_i}	21	22	17	18	16	15	17	23	26	23	15	21	20	19	22	18	14	23	23	10	12	16	21	9

5.4.2. Detection Phase

In the detection phase, we monitor the running of programs and compute the probability of their corresponding action sequences by applying Algorithm 4. In the experiments, we assume the rejection rate $fracrej = 0.05$ in Algorithm 3 and, by default, we use program behavior-aware topology to train the HMMs, λ_{c_i}.

The impact of clustering on the performance of HM^3alD

When k, the number of clusters, increases in the clustering process of HM^3alD, the corresponding reconstruction error decreases and also the number of clusters is increased, so it means we should learn more HMMs. It means we should trade off between k and the set of {DR, FAR, Accuracy} metrics derived from corresponding HMMs, $(\lambda_{c_i}, \mathcal{T}_{c_i})$. Figures 5–7 show the average and STD (Standard deviation) of DR, FAR, and Accuracy of HM^3alD for different candidate values of k, based on Figure 4 respectively.

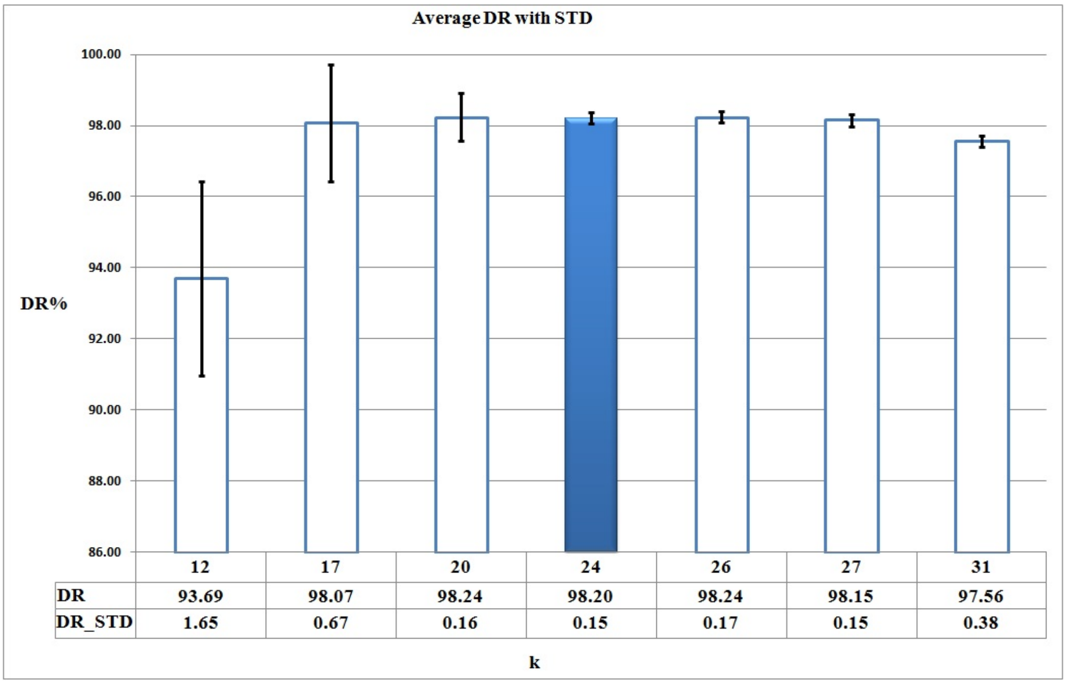


Figure 5. DR for different candidate values of k based on Figure 4.

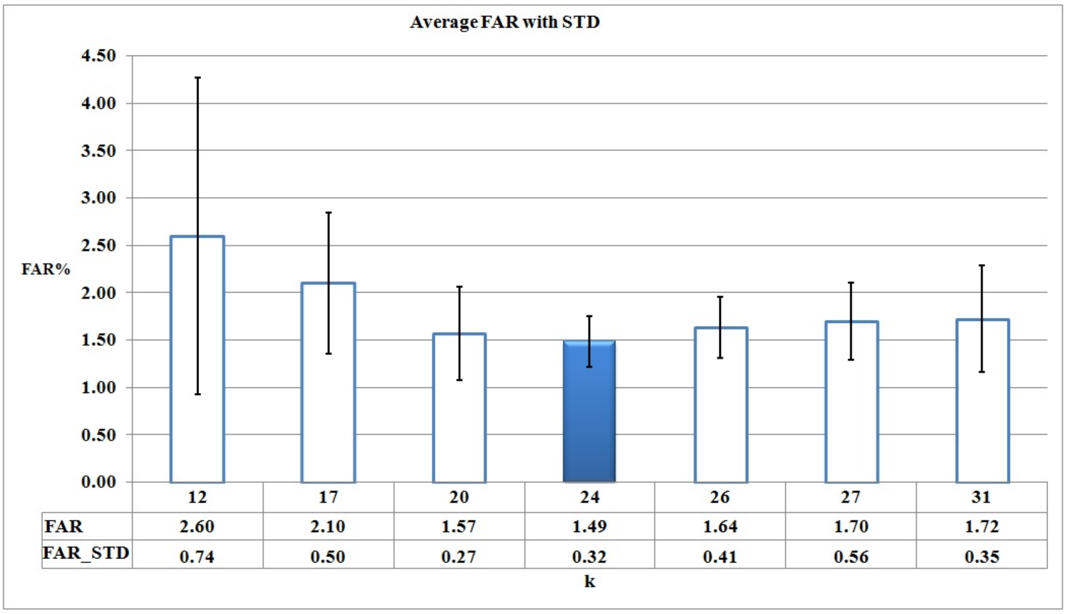


Figure 6. FAR for different candidate values of k based on Figure 4.

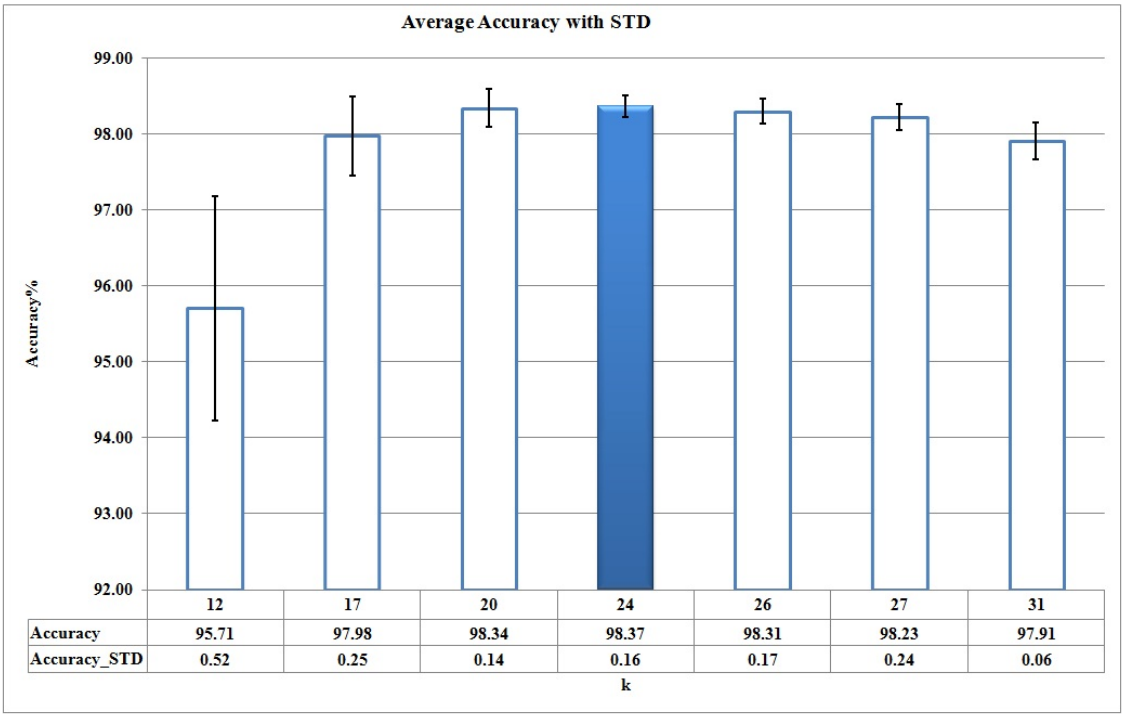


Figure 7. Accuracy for different candidate values of k based on Figure 4.

As it can be seen in Figures 5–7, DR, FAR, and Accuracy metrics for $k = 24$ are better in comparison with the results for other k values. This happens because, when considering a small number of clusters (for example k = 3, 6, 12), malware action sequences that are less similar are assigned to the same cluster, so no suitable HMMs are learned. Thus, the learned HMMs are neither able to detect malware programs properly nor discriminate between malware and benign programs as good as possible. On the other hand, the large number of clusters increases the complexity of HM^3alD and causes each cluster to have too few action sequences (for example, k = 26, 27, 31).

Cross-validation results show that 19 of 24 malware HMMs (i.e., 80% of the learned HMMs) compute their decision thresholds based on the maximum benign probability Pb_{c_i} in Algorithm 3. This fact means that HMMs are trained suitably and effectively discriminate between malware and benign action sequences.

The impact of program behavior-aware topology

Figure 8 shows the comparison results of HM^3alD and HM^3alD_R (without program behavior-aware topology). As it is seen, HM^3alD_R dramatically increases FAR. In other words, HM^3alD that preserves program behavior-aware topology, effectively reduces FAR 227%, which is hard work without any impact on DR.

The impact of expressing system call sequences as action sequences

To explain the importance of the action sequences, we examine the proposed method by considering the raw system call sequences (called HM^3alD_S) instead of the action sequences. $HM^3alD\_S\_R$ denotes the random topology version of HM^3alD_S. Recall that it is shown that methods based on raw system call sequences produce many false positives [39], and not considering their parameters such as frequency and arguments would result in high false alarm rate [40,41]. Figure 9 shows the efficiency of HM^3alD in comparison with HM^3alD_S and $HM^3alD\_S\_R$ in terms of FAR and DR. It is seen that in HM^3alD, FAR is decreased 453%, and DR is increased 5.8%. As an important result, using action sequences in HM^3alD achieves high DR and low FAR, that is a difficult task in HMM methods as a one-class classifier approach.

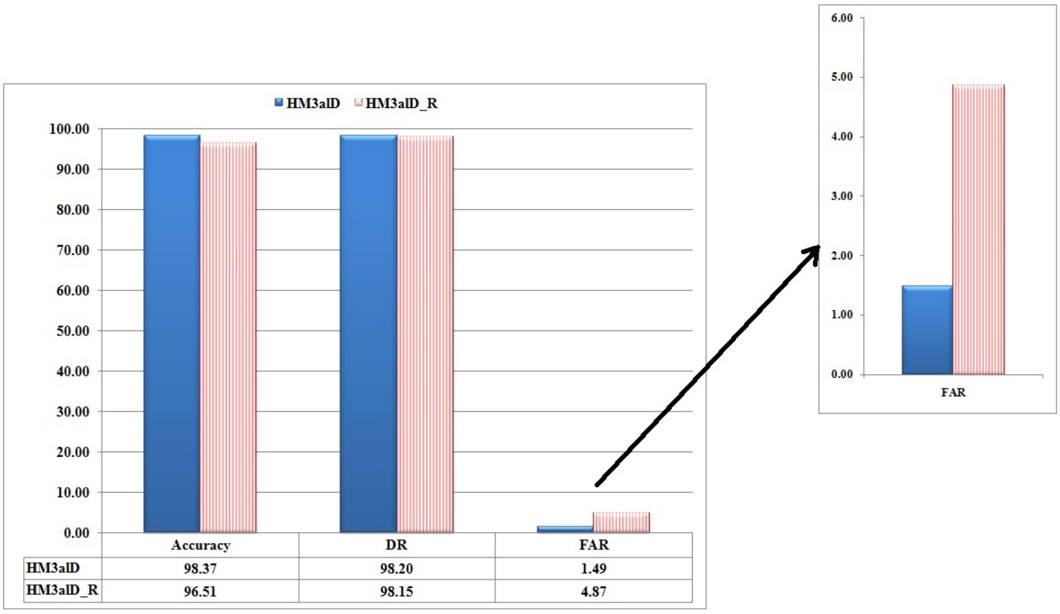


Figure 8. The performance comparison of HM^3alD with and without the program behavior-aware topology for $k = 24$ and $fracrej = 0.05$.

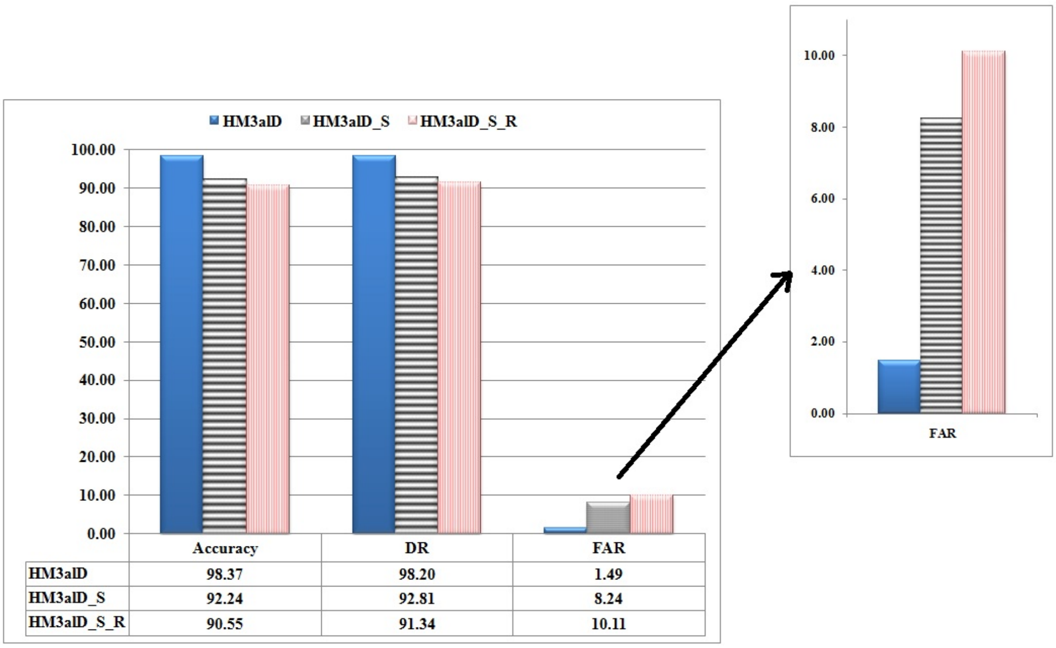


Figure 9. The performance comparison of HM^3alD (for $k = 24$), HM^3alD_S (for $k = 26$) and $HM^3alD\_S\_R$ (for $k = 26$).

5.5. Comparing with Other Work

In this section, the performance of HM^3alD in terms of DR, FAR, and Accuracy is compared to important previous works. For fair comparison, we consider the number of dataset members same as the compared methods when we train HMMs of HM^3alD. To express the detection performance of HM^3alD, we compare HM^3alD to dynamic malware detection methods and show the corresponding results in Table 4. Moreover, Table 5 shows the results of HM^3alD in comparison with static malware detection methods. As shown in Tables 4 and 5, HM^3alD outperforms all dynamic and static malware detection methods especially in term of FAR that is a hard work to decrease it without affecting DR.

Table 4. Comparison of HM^3alD with current dynamic malware detection methods.

Approaches	Dataset Size (Benign/Malware)	DR(%)	FAR(%)	Accuracy (%)
Shahzad (2013) [20]	105/114	93.7	0	96.65
HM^3alD	105/114	**100**	**0**	**100**
Elhadi (2014) [21]	98/416	97.57	0	98.05
HM^3alD	98/416	**100**	**0**	**100**
Salehi (2017) [23]	1359/3009	98.4	4.6	—
HM^3alD	1359/3009	**98.89**	**1.12**	**98.88**
Shehata (2015) [22]	2000/2000	97.6	2.37	96.89
HM^3alD	2000/2000	**98.83**	**1.18**	**98.87**

Table 5. Comparison of HM^3alD with the current static malware detection methods.

Methods	Dataset Size (Benign/Malware)	DR(%)	FAR(%)	Accuracy (%)
Kalbhor (2015) [13]	370/760	88.95	**0.2**	97.58
HM^3alD	370/760	**99.12**	0.23	**99.38**
Song (2012) [16]	8/200	100	12.5	99.52
HM^3alD	8/200	**100**	**0**	**100**
Shahid (2015) [5]	2330/1020	98.9	4.5	—
HM^3alD	2330/1020	**98.99**	**1.27**	**98.85**
Furuki (2012) [12]	2595/3639	98.4	2.7	97.85
HM^3alD	2595/3639	**98.81**	**1.36**	**98.70**
Ding (2013) [17]	3760/4410	97.3	—	91.2
HM^3alD	2676/4410	**98.79**	**1.85**	**98.39**

Expressing the behavior of a malware program leads to detect its polymorphic instances, effectively. HM^3alD tries to extract realistic behavior of malware from polymorphic instances. First, HM^3alD abstracts malware behavior by using high-level action sequences instead of system call sequences. Then, it clusters similar malware action sequences leading to integrate polymorphic instances of a malware. Finally, HM^3alD models the realistic behavior of each malware cluster using HMM. Thus, HM^3alD provides a high detection rate and low false rate in comparison with other work.

Note that a false positive occurs when HM^3alD erroneously labels a benign program as malware. If a malware detector blocks access or deletes a program or file that is vital to the proper functioning of some system programs, those may become unusable and, in some cases, the deletion may render a system unstable, although the anti-malware methods try to reduce FAR to zero. For Virus Bulletin [42], "the 'no false positives' rule is one of the main requirements for certification in the VB100 test process". AV-Comparatives [43] considers false positives "an important measurement for AV quality", and an important factor in determining the reliability of a product, besides its detection capabilities. Thus, we introduce new metric, $PenalizedDR$, as Equation (7).

$$PenalizedDR = DR - \alpha * FAR, \tag{7}$$

We analyze the impact of α by increasing it from 0 to 10 in increments of 0.5. Figures 10 and 11 show the results of $PenalizedDR$ for HM^3alD and current dynamic and static methods, respectively.

The important findings from the results shown in Figures 10 and 11 are as follows: (1) FAR is the most important and impressive measurement for malware detection methods; (2) high DR is not enough to have a good result and its effectiveness depends on FAR; and (3) high FAR significantly reduces the usefulness and applicability of such methods. According to Tables 4 and 5, and Figures 10 and 11, the performance of HM^3alD is always higher than the other methods in terms of DR and Accuracy and is significantly better than the other methods in term of FAR.

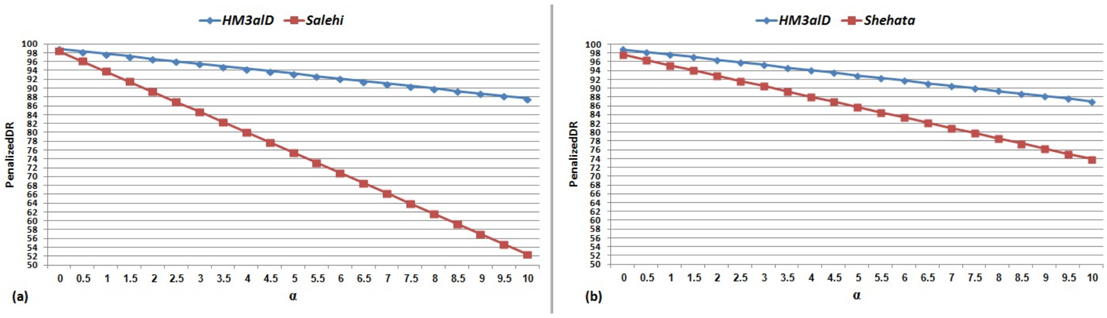


Figure 10. $PenalizedDR$ for HM^3alD and current dynamic methods: (**a**) Salehi (2017) [23]; and (**b**) Shehata (2015) [22].

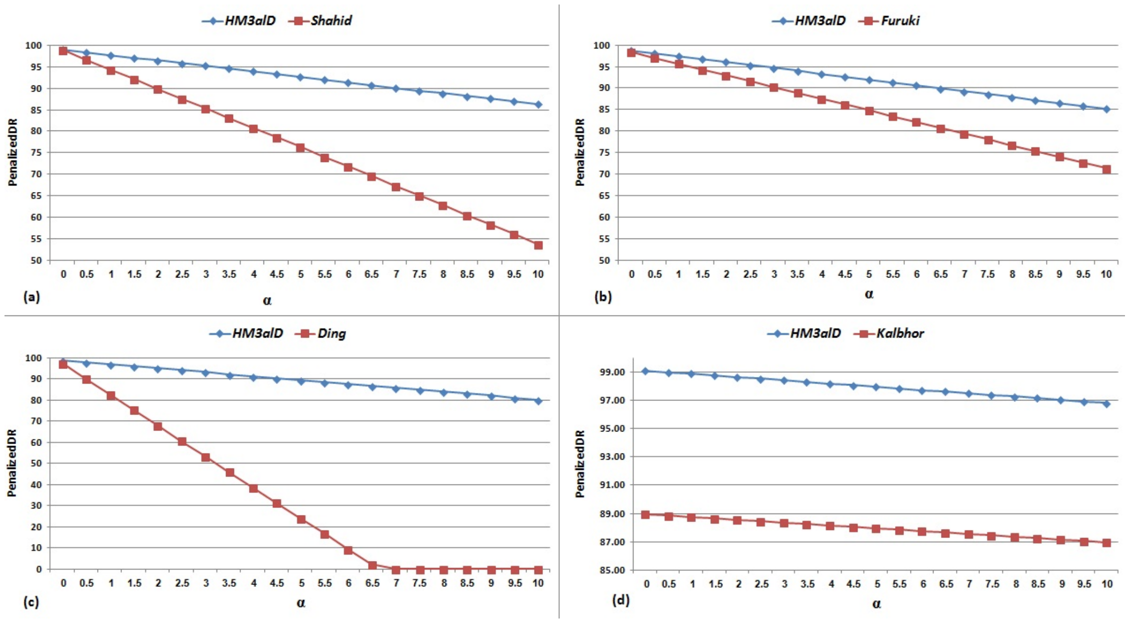


Figure 11. $PenalizedDR$ for HM^3alD and current static methods: (**a**) Shahid (2015) [5]; (**b**) Furuki (2012) [12]; (**c**) Ding (2013) [17]; and (**d**) Kalbhor (2015) [13]. Note that we estimated FAR value of Ding work [17] from DR and Accuracy.

6. Analysis and Discussion

We analyze the proposed method, HM^3alD, from three aspects: action sequences, HMM topology, and the impact of clustering and decision thresholds on learned HMMs.

Expressing the realistic behavior of a malware program is definitely effective in detecting its polymorphic instances. Therefore, it is crucial to have an effective yet feasible way to express malware behavior. According to experimental results, raw data from malware at runtime that is merely a sequence of system calls is too fine-grained to be helpful in expressing the realistic behavior of a suspicious program. Therefore, HM^3alD defines and operates on some high-level "actions" (i.e., read file, write file, send data, and remove a registry key) where a sequence of these actions represents a more meaningful behavior. Results shown in Figure 9 indicate that, without a high level action sequence, we would not find acceptable performance for the proposed method especially in term of FAR.

In HMM, designing HMM topology is really important. Taking this fact into account, HM^3alD employs a special HMM topology to learn malware behaviors in coarse grain. The resulting HMM topology is built based on a triple-stage (initialization, running, and termination) execution of a program. This approach has a great impact on training and building suitable HMMs and, compared to the other similar methods (e.g., [13]), HM^3alD provides a drastic decrease in FAR without any alteration in DR. As Figure 8 shows, if we use a random topology rather than a program behavior-aware topology in the training phase, the learned HMMs are not much effective.

Furthermore, a key feature of HM^3alD is that we train each HMM using only malware action sequences (i.e., without using any benign action sequences) that leads to reducing the complexity of HM^3alD dramatically, in the training phase. Note that, in computing the HMM decision thresholds, we use malware and a few benign action sequences.

Clustering action sequences makes it possible to put similar action sequences in the same group. Results indicate that without proper clustering of action sequences, we cannot train suitable HMMs in the training phase, which leads to unacceptable malware detection results.

Our implementation of HM^3alD shows great promise in DR and especially in FAR even when the number of programs goes beyond the capability of some current methods.

7. Conclusions and Future Work

In this study, we showed that it is possible to employ HMMs to detect polymorphic malware in a dynamic manner. In this paper, we propose a novel dynamic detection method, named HM^3alD, based on HMM to detect polymorphic malware. We show that the proposed method could effectively distinguish between benign and malware programs. HM^3alD was trained only by malware action sequences to detect polymorphic malware on the host side at runtime.

In the future, we will work on the applications of this method in classifications of malware programs, detecting anomalies and generating behavioral signatures. We will also try to perform the detection at the early stages of program execution, as much as possible.

Author Contributions: A.T. is the author who mainly contributed to this research, performing experiments, and writing the manuscript. S.J. read and approved the final manuscript, and reviewed the results.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Mapping a Subsequence of System Calls to Actions

High-level actions derived from subsequences of system calls are classified into seven sets:

1. File Actions include “write file”, “read file”, “delete file”, “execute file”, “copy file”, and “move file”.
2. Registry Actions indicate the program behavior regarding the registry of Windows operating system (OS), which includes writing, reading, and deleting in the registry.
3. Service Actions relate to the registered services in Windows OS. These actions include creating, deleting, and executing a service in Windows.
4. Network Actions cover the behavior of executing sample in the transport layer. This Action set is formed based on the state diagram in the TCP protocol, which includes opening and closing a connection (socket), listening on a socket, binding a socket, accepting a socket, sending and receiving on a socket, etc.
5. Internet Actions include all actions that occur during communications of a running program in the Application Layer. These actions include opening a session, session connection, and sending and reading files via the network.
6. System Actions consist of the sequence of system calls which express a system operation to begin the execution. Load library, memory allocation, address allocation to procedures, etc. are some examples of system actions.
7. Process Actions include all the actions related to process and threads, (e.g., creating, executing, and killing a thread.)

For example, we map the subsequence of system call {NTCreateFile, NTOpenFile, NTReadFile, NTReadFile, NTReadFile, NTCloseFile} to the action *ReadFile*.

Appendix B. The Forward Algorithm

Algorithm 5 : Forward algorithm

It implements the scaled forward algorithm and returns:
(1) log($P(O|\lambda)$); (2) scale factor set $F = \{f_1, f_2, \ldots, f_T\}$; and (3) scale forward variables, $\hat{\alpha}_t(i)$.

Input:

O = an induced action sequence $\{O_1, O_2, \ldots, O_T\}$ corresponding to a program P
A = state transition probability matrix with the number of states (N)
B = observation probability matrix

Outputs:

scaled forward value = $logP(O|\lambda)$
scale factor set $F = \{f_1, f_2, \ldots, f_T\}$
scale forward variables, $\hat{\alpha}_t(i)$ for all $i = 1 \ldots N$ and $t = 1 \ldots N$

Initialization:
$f_1 = 0$
for $i = 1$ to N **do**
    $\alpha_1(i) = \pi_i b_i(O_1)$
    $\alpha'_1(i) = \alpha_1(i)$
    $f_1 = f_1 + \alpha_1(i)$
end for
$f_1 = \frac{1}{f_1}$
for $i = 1$ to N **do**
    $\hat{\alpha}_1(i) = f_1 \alpha_1(i)$
end for
Induction:
for $t = 2$ to T **do**
    $f_t = 0$
    for $i = 1$ to N **do**
        $x = 0$
        for $j = 1$ to N **do**
            $x = x + \hat{\alpha}_{t-1}(j) a_{ji}$
        end for
        $\alpha'_t(i) = b_i(O_t) x$
        $f_t = f_t + \alpha'_t(i)$
    end for
    $f_t = \frac{1}{f_t}$
    for $i = 1$ to N **do**
        $\hat{\alpha}_t(i) = f_t \alpha'_t(i)$
    end for
end for
Termination
$log\_p = 0$
for $t = 1$ to T **do**
    $log\_p = log\_p + log(f_t)$
end for
$log\_p = -log\_p$
return log_p

References

1. Kang, B.; Han, K.S.; Kang, B.; Im, E.G. Malware categorization using dynamic mnemonic frequency analysis with redundancy filtering. *J. Digit. Investig.* **2014**, *11*, 323–335. [CrossRef]
2. Shin, S.; Zhaoyan, X.; Guofei, G. EFFORT: A new host–network cooperated framework for efficient and effective bot malware detection. *Comput. Netw.* **2013**, *57*, 2628–2642. [CrossRef]
3. Symantec. Available online: http://www.symantec.com/threatreport/ (accessed on 25 June 2018).
4. Lu, H.; Wang, X.; Zhao, B.; Wang, F.; Su, J. ENDMal: An anti-obfuscation and collaborative malware detection system using syscall sequences. *J. Math. Comput. Model.* **2013**, *58*, 1140–1154. [CrossRef]
5. Shahid, A.; Horspool, R.N.; Traore, I.; Sogukpinar, I. A framework for metamorphic malware analysis and real-time detection. *J. Comput. Secur.* **2015**, *48*, 212–233.
6. Cha, S.K.; Moraru, I.; Jang, J.; Truelove, J.; Brumley, D.; Andersen, D.G. SplitScreen: Enabling efficient, distributed malware detection. *J. Commun. Netw.* **2011**, *13*, 187–200. [CrossRef]
7. Egele, M.; Scholte, T.; Kirda, E.; Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.* **2012**, *44*, 6. [CrossRef]
8. Bruschi, D.; Martignoni, L.; Monga, M. Code normalization for self-mutating malware. *IEEE J. Secur. Priv.* **2007**, *5*, 46–54. [CrossRef]
9. Chandola, V.; Banerjee, A.; Kumar, V. Anomaly detection: A survey. *J. ACM Comput. Surv.* **2009**, *41*, 15. [CrossRef]
10. Grill, M.; Pevny, T. Learning combination of anomaly detectors for security domain. *Comput. Netw.* **2016**, *107*, 55–63. [CrossRef]
11. Preda, M.D.; Christodorescu, M.; Jha, S.; Debray, S. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.* **2008**, *30*, 25. [CrossRef]
12. Faruki, P.; Laxmi, V.; Gaur, M.S.; Vinod, P. Mining control flow graph as API call-grams to detect portable executable malware. In Proceedings of the ACM International Conference on Security of Information and Networks, Jaipur, India, 25–27 October 2012.
13. Kalbhor, A.; Austin, T.H.; Filiol, E.; Josse, S.; Stamp, M. Dueling hidden Markov models for virus analysis. *J. Comput. Virol. Hacking Tech.* **2015**, *11*, 103–118. [CrossRef]
14. Wong, W.; Stamp, M. Hunting for metamorphic engines. *J. Comput. Virol.* **2006**, *2*, 211–229. [CrossRef]
15. Austin, T.H.; Filiol, E.; Josse, S.; Stamp, M. Exploring hidden markov models for virus analysis: A semantic approach. In Proceedings of the 46th Hawaii International Conference on System Sciences, Wailea, HI, USA, 7–10 January 2013.
16. Song, F.; Touili, T. Efficient malware detection using model-checking. *J. Formal Methods (FM)* **2012**, *7436*, 418–433.
17. Ding, Y.; Yuan, X.; Tang, K.; Xiao, X.; Zhang, Y. A fast malware detection algorithm based on objective-oriented association mining. *J. Comput. Secur.* **2013**, *39*, 315–324. [CrossRef]
18. Hellal, A.; Lotfi, B.R. Minimal contrast frequent pattern mining for malware detection. *J. Comput. Secur.* **2016**, *62*, 19–32. [CrossRef]
19. Park, Y.; Reeves, D.S.; Stamp, M. Deriving common malware behavior through graph clustering. *J. Comput. Secur.* **2013**, *39*, 419–430. [CrossRef]
20. Shahzad, F.; Shahzad, M.; Farooq, M. In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS. *J. Inf. Sci.* **2013**, *231*, 45–63. [CrossRef]
21. Elhadi, A.A.E.; Maarof, M.A.; Barry, B.I.; Hamza, H. Enhancing the detection of metamorphic malware using call graphs. *J. Comput. Secur.* **2014**, *46*, 62–78. [CrossRef]
22. Shehata, G.H.; Mahdy, Y.B.; Atiea, M.A. Behavior-based features model for malware detection. *J. Comput. Virol. Hacking Tech.* **2015**, *12*, 59–67.
23. Salehi, Z.; Sami, A.; Ghiasi, M. MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values. *J. Eng. Appl. Artif. Intell.* **2017**, *59*, 93–102. [CrossRef]
24. Christodorescu, M.; Jha, S.; Kruegel, C. Mining specifications of malicious behavior. In Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 3–7 September 2007.

25. Kolbitsch, C.; Comparett, P.M.; Kruegel, C.; Kirda, E.; Zhou, X.; Wang, X. Effective and efficient malware detection at the end host. In Proceedings of the International Conference on USENIX Security Symposium, Montreal, QC, Canada, 10–14 August 2009.
26. Ding, Y.; Xiaoling, X.; Sheng, C.; Ye, L. A malware detection method based on family behavior graph. *J. Comput. Secur.* **2018**, *73*, 73–86. [CrossRef]
27. Schrittwieser, S.; Stefan, K.; Johannes, K.; Georg, M.; Edgar, W. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.* **2016**, *49*, 4:1–4:37. [CrossRef]
28. Cohen, Y.; Danny, H. Scalable Detection of Server-Side Polymorphic Malware. *Knowl.-Based Syst.* **2018**, *156*, 113–128. [CrossRef]
29. Alpaydin, E. *Introduction to Machine Learning*; MIT Press: London, UK, 2014.
30. Camastra, F.; Vinciarelli, A. *Machine Learning for Audio, Image, and Video Analysis*; Springer: London, UK, 2008.
31. Francis, H.; Chen, H.; Ristenpart, T.; Li, J.; Su, Z. Back to the future: A framework for automatic malware removal and system repair. In Proceedings of the 22nd Annual Computer Security Applications Conference, Miami Beach, FL, USA, 11–15 December 2006.
32. Cuckoo Foundation. Available online: https://www.cuckoosandbox.org (accessed on 25 June 2018).
33. Murtagh, F. A survey of recent advances in hierarchical clustering algorithms. *Comput. J.* **1983**, *26*, 354–359. [CrossRef]
34. Yujian, L.; Bo, L. A normalized Levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.* **2007**, *29*, 1091–1095. [CrossRef] [PubMed]
35. VxHeavens. Available online: http://vxheaven.org/vl.php (accessed on 1 March 2016).
36. Sourceforge. Available online: https://sourceforge.net/ (accessed on 25 June 2018).
37. GHMM Library. Available online: http://www.ghmm.org/ (accessed on 25 June 2018).
38. Giacinto, G.; Perdisci, R.; Del Rio, M.; Roli, F. Intrusion detection in computer networks by a modular ensemble of one-class classifiers. *J. Inf. Fusion* **2008**, *9*, 69–82.
39. Lanzi, A.; Balzarotti, D.; Kruegel, C.; Christodorescu, M.; Kirda, E. Accessminer: Using system-centric models for malware protection. In Proceedings of the 17th ACM conference on Computer and communications security, Chicago, IL, USA, 4–8 October 2010; pp. 399–412.
40. Das, S.; Liu, Y.; Zhang, W.; Chandramohan, M. Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware. *IEEE Trans. Inf. Forensics Secur.* **2016**, *11*, 289–302. [CrossRef]
41. Maggi, F.; Matteucci, M.; Zanero, S. Detecting intrusions through system call sequence and argument analysis. *IEEE Trans. Dependable Secur. Comput.* **2010**, *7*, 381–395. [CrossRef]
42. Virus Bulletin. Available online: https://www.virusbulletin.com (accessed on 25 June 2018).
43. AV-Comparatives. Available online: http://www.av-comparatives.org/false-alarm-tests (accessed on 1 September 2016).