Employing Program Semantics for Malware Detection

Smita Naval, Vijay Laxmi, *Member, IEEE,* Muttukrishnan Rajarajan, *Senior Member, IEEE,* Manoj Singh Gaur, *Member, IEEE,* and Mauro Conti, *Senior Member, IEEE*

***Abstract*—In recent years, malware has emerged as a critical security threat. Additionally, malware authors continue to embed numerous anti–detection features to evade existing malware detection approaches. Against this advanced class of malicious programs, dynamic behavior–based malware detection approaches outperform the traditional signature–based approaches by neutralizing the effects of obfuscation and morphing techniques. The majority of dynamic behavior detectors rely on system–calls to model the infection and propagation dynamics of malware. However, these approaches do not account an important anti–detection feature of modern malware, *i.e.*, system–call injection attack. This attack allows the malicious binaries to inject irrelevant and independent system–calls during the program execution thus modifying the execution sequences defeating the existing system–call based detection. To address this problem, we propose an evasion–proof solution that is not vulnerable to system–call injection attacks. Our proposed approach precisely characterizes the program semantics using Asymptotic Equipartition Property (AEP) mainly applied in information theoretic domain. The AEP allows us to extract the information–rich call sequences that are further quantified to detect the malicious binaries. Furthermore, the proposed detection model is less vulnerable to call–injection attacks as the discriminating components are not directly visible to malware authors. This particular characteristic of proposed approach hampers a malware author's aim of defeating our approach. We run a thorough set of experiments to evaluate our solution and compare it with existing system-call based malware detection techniques. The results demonstrate that the proposed solution is effective in identifying real malware instances.**

***Index Terms*—Malware, Malware Detection, System–calls, Semantically–relevant paths, System–call injection attacks**

I. Introduction

OVER the last decade, malware has emerged as a crucial security threat. The proliferation of advanced computing and networking technology has empowered malware programs with advanced anti-detection and anti-analysis features. The advanced malware programs instigate a variety of attacks such as Distributed Denial of Service (DDoS) attacks, social engineering attacks and clickfraud attacks, to name a few. Malware is a persistent threat to any computer system's integrity, confidentiality, and availability [1]. These software programs have a disruptive impact on our applications, service providers, storage, servers, and networks. In 2013, AV-test institute discovered a total of 100 million new malicious files and this number has reached 120 million in 2014 [2]. This explosion of completely new malware threats and variants of existing malicious threats cause substantial damage in terms of financial losses. For instance, Stuxnet, Rocra, Code-Red, and Slammer are few known malware threats that induced significant financial losses costing billions of US dollars [3]. This trend is continuing and requires an assurance to mitigate these malicious threats. Towards the security and privacy of connected systems, malware detection becomes the first line of defense.

A solution that can detect almost every malicious program is practically impossible [4], and the evidence of this belief can be seen by the detection efficiency of existing Anti-Virus (AV) products [5]. These AV solutions mostly depend on signature databases that need to be updated frequently. While the process of malware creation has advanced, the malware detection process still relies on signature-based approaches and thus has been proved to be inefficient in capturing new and advanced class of malware samples.

Existing arsenal of malware detection solutions relies on static and dynamic techniques. The static techniques look for syntactic markers or signatures to detect malware. Analyzing a malware sample to identify the unique static markers requires greater efforts as the effectiveness of these markers is hampered by obfuscation and morphism (polymorphism and metamorphism) techniques. As a consequence, the static methods for malware detection may not capture unknown malware instances [6]. To nullify the effects of obfuscation, polymorphism and metamorphism on malware executables, researchers have given preference to dynamic malware detection approaches. In particular, the dynamic behavior-based malware detection approaches utilize the semantic of a malware program by examining its runtime interaction with system objects, resources, and services [7]. Therefore, these approaches are well suited for capturing new and syntactically different but semantically similar unseen malware variants.

The majority of dynamic behavior-based malware detectors [8]–[11] makes use of system-calls as these provide an interface of application's interaction with Operating System (OS). A system call is an interface between a user-level application and kernel-level services. These services include hardware, input-output related activities, creation/deletion of processes and many more. To infect the host system, malware needs to invoke a sequence of system-calls as these are nonby-

S. Naval[1], V. Laxmi[2] and M. S. Gaur[3] are with the Department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur-302017, India E-mail: (smita.710@gmail.com[1], vlaxmi@mnit.ac.in[2], gaurms@mnit.ac.in[3])

M. Rajarajan is with the Department of Security Engineering, City University London, Northampton Square, London, UK E-mail: r.muttukrishnan@city.ac.uk

M. Conti is with the Department of Mathematics, University of Padua, Padua 35131, Italy E-mail: conti@math.unipd.it

passable. Therefore, capturing malware by employing system-calls will allow devising a reliable detection solution. However, the present malware programs are equipped with advanced anti-detection techniques which can evade even system-call based malware detectors [12].

To counter system-call based approaches, malware authors make use of shadow attacks [13] and system-call injection attacks [14]. The feasibility of former attacks was first demonstrated by authors in [13]. The authors in their paper have shown that the critical system-call sequences of malware can be divided and exported into separate shadow processes. The shadow processes individually act in benign manner and collectively these depict malicious behavior. These shadow processes communicate with rewritten malicious code to deliver their malicious payload. The system-call injection attacks are deployed by inserting irrelevant and independent calls in the actual execution flow of malware binaries. By doing so, detection approaches based on graph matching or path similarity analysis are defeated. These attacks are the variant of code-injection attacks [14], [15]. The shadow attacks suffer from the following limitations that restrict its applicability in practice.

1) The shadow attacks lead to multi-process malware that is slower than the original single process malware. Such a malware, cannot be used in various real-time attack situations (such as chain attacks) [16].

2) The implementation of these attacks requires the division of malware; the communication of multiple processes; the bootstrap, and the execution sequence of multiple processes. Failure of any shadow process will result into the failure of entire process [17].

The aforementioned challenges limit the feasibility of the shadow attacks. On the other hand, the system-call injection attacks are free of these limitations, and, therefore, to earn more revenue, malware authors would prefer these attacks. Taking this fact into consideration, we present an effective system-call based malware detection approach that is resistant against system-call injection attacks.

In this paper, we devise a novel malware detection mechanism, which is resilient against system-call injection attacks. The proposed detector characterizes the program behavior by exploiting the system-level information flow. The characterized behavior is represented in terms of semantically-relevant paths employed to build and train the feature space. For extracting and identifying semantically-relevant paths, we adopt the concept of Asymptotic Equipartition Property (AEP [18]) from information theory. According to AEP, in any graph, there exists a few paths that carry almost all information of the graph. Following this concept and the proofs given in [19], we apply AEP in our approach to extract semantically-relevant paths, which depict the program behavior. We construct a set of such semantically-relevant paths on which we apply a measure called Average Logarithmic Branching Factor (ALBF) [19] to build our feature space. Finally, our model is trained to differentiate between benign and malware programs. The contributions of this paper are summarized below:

- We propose a novel malware detection approach that characterizes the program behavior in terms of semantically-relevant paths. In our approach, these paths are extracted by exploiting system-level information flow.
- We form a novel feature space constructed by quantifying the semantically-relevant paths using ALBF metric. Unlike other approaches [20]–[22], our feature space consists of non-string features, and, therefore, it makes our method an evasion-proof solution to malware authors.
- We show that our approach is resilient against system-call injection attacks. The performance of our model remains consistent even after injecting thousands of independent system-calls into malware traces.
- We have tested the detection capability of our model with real benign and malware instances. We observe the detection accuracy of $\sim$95% which demonstrates the effectiveness of our model in identifying real malicious attacks.

The remainder of the paper is organized as follows: Section II states our problem. The proposed method is illustrated in Sections III and IV. The experimental setup that includes dataset collection and results is discussed in Section V. In Section VI, we discuss the merits and demerits of our approach. Section VII highlights the existing work in the domain of malware detection. Finally, the concluding remarks are given in Section VIII.

II. The Problem

To evade detection, malware is continuously being evolved and equipped with anti-detection techniques such as code obfuscation, polymorphism, metamorphism, anti-debugging, anti-VM, code-injection, to name a few. By incorporating these techniques into malicious code, malware authors try to extend the lifetime of malicious code and hide its actual malicious intent. The anti-detection techniques have instigated a never-ending arms-race between malware detectors and malware authors. In this paper, we propose a novel detection technique for identifying detection-aware malicious threats. Our proposed approach investigates the program semantics to identify the information-rich components of malware and benign files.

A. Technical Viewpoint

The Complexity of sophisticated malware codes makes them difficult to detect and analyze. These programs can be found in multiple statically diverse forms having the same functionality. Therefore, to understand the behavior of a malware program we have to dive into its semantics instead of the syntax. The semantic (behavior) of any program can be explored by exploiting its execution-flow. During execution, the malicious programs try to infect host machine with actual malicious payload (if it is not environment-aware [23]–[25] or having trigger-based behavior [26]). Our prime objective is to define a metric that can quantify the program semantics. For this, we consider AEP that is based on Shannon's entropy [27]. The Shannon's entropy describes the amount of meaningful information present in a program object [19] as it remains

unchanged when one-to-one function is applied [18]. Using entropy, we extract semantically-relevant call sequences and quantify them to construct feature space of the proposed detection model. Our notion of characterizing program semantics is not vulnerable to call-injection attack or behavior obfuscation as the discriminating components are composed of: 1) multiple call sequences, and 2) non-string based features.

The behavior components of malware often exhibit similar behavior with the different set of call sequences incurred due to the injection attacks. For example, self-replication behavior of malware in which it copies its content to either a new file or into an already existing file, can be represented in one of the following system-call based path sequences: 1) `NtCreateFile`→`NtOpenFile`→`NtReadFile`→`NtWriteFile`, 2) `NtOpenFile`→`NtCreateFile`→`NtReadFile`→`NtWriteFile`, and 3) `NtCreateSection`→`NtMapViewOfSection`→`NtCreateFile`→`NtSetInformationFile`→`NtWriteFile`. To capture these paths showing similar behavior, we represent the program behavior through multiple paths carrying almost the same information quotient. These behavior components are further transformed into a non-string based feature space to avoid string-based evasions [1]. The problem statement is composed of sub-problems listed as follows:

1) Encapsulating program behavior into semantically-relevant paths through AEP concept and extract them via ALBF.
2) Verify and validate the specified behavior by constructing a learning-based detection model.

The proposed approach enables us to transform the input binary programs into two forms; one that characterizes the most relevant information; andthe other that exploits this relevant information to construct a detection model and thus verifies effectiveness of extracted behavior.

III. Encapsulating Program Behavior

In this phase, execution traces of binaries are transformed into Ordered System-Call Graph (OSCG) derived from the sequence of invoked system-calls. A vertex of OSCG corresponds to a system-call in program trace. An edge from vertex u to vertex v of OSCG corresponds to the occurrence of the pair $\langle S_u, S_v \rangle$ in the sequence. Here, S_u and S_v are system-calls corresponding to vertices u and v respectively. The graph preserves order. So, a pair $\langle S_1, S_2 \rangle$ shall add an edge from vertex 1 to 2, whereas $\langle S_2, S_1 \rangle$ shall add an edge from vertex 2 to 1. An OSCG is constructed for each input binary. In order to specify program behavior, the OSCGs are used to determine all reachable paths from initial node (the first call invoked) to the final node (last call invoked) of the sample. We apply AEP on each path to check if it is semantically-relevant path. The detailed description is given in subsequent paragraphs.

A. Transforming Program Binaries as OSCG

To transform binaries into ordered system-call graph (OSCG), each binary is executed in a virtualized environment. In particular, we have employed Ether [28]. Execution of binaries is monitored and invoked system-calls are logged. We prefer Ether to other analysis frameworks as it provides host-based tracing by employing hardware virtualization. It is resilient to anti-debugging, anti-emulation and code-obfuscation and in-guest changes are also made hidden [29]. Ether produces a page fault or exception to intercept the system-calls made by the target application. Whenever this application requires a system service, it executes `SYSENTER` that transfers the control to kernel space where it copies the value (address) stored in a special register `SYSENTER_EIP_MSR` into instruction pointer (`IP`). Ether sets `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 `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.

The acquired traces are used in extracting the sequence of invoked system-calls. Consider an execution trace $\xi = S_1, S_2, S_3, S_1, S_2, S_2, S_3, S_3, S_2$. This trace has three distinct system-calls S_1, S_2 and S_3. So, we construct OSCG with three nodes. As the sequence has pairs $\langle S_1, S_2 \rangle, \langle S_2, S_3 \rangle, \langle S_3, S_1 \rangle$ and $\langle S_3, S_2 \rangle$, edges are added from node 1 to 2, node 2 to 3, node 3 to 1, and node 3 to 2. Graph-based representation such as OSCG, also, captures the sequential nature of the data [30]. Representing execution traces in the form of directed labeled graph is not new. In the past, many approaches have used graph-based representations to detect malicious files [8], [9], [11], [31]. In OSCG, we ignore all the system-call parameters to avoid the sensitivity towards handles, arguments and other system artifacts.

We shall be using ξ to represent execution trace of a sample and $\mathbb{S}$ to represent the set of all possible (distinct) system-calls. In our case $|\mathbb{S}| = 284$, *i.e.*, $\mathbb{S} = \{S_1, S_2, \cdots, S_{284}\}$ as only 284 possible system-calls can be invoked on Windows XP (SP2) [32]. Each call in $\mathbb{S}$ performs a service at the kernel level that is requested by running binary. For instance, routine `NtMapViewOfSection` is only invoked to map the view of a section into the virtual address space of running process, `NtWriteFile` is called to write into a file and `NtClose` is the routine invoked to close the handles created by other routines. Successive calls to these routines collectively depict program behavior. To characterize the program behavior through OSCG, we preserve the order in which system-calls are invoked.

Definition 1 An Ordered System-Call Graph (OSCG) $\mathbb{G} = (\mathbb{S}, \mathbb{E})$ is a directed graph, where $\mathbb{S}$ is the set of vertices and each vertex represents a system-call. $\mathbb{E} = \{E_{ij} | S_i \xrightarrow{\rho_{ij}} S_j;\ S_i, S_j \in \mathbb{S}\}$, where ρ_{ij} denotes the transition probability from system-call S_i to system-call S_j.

It is assumed that paths in graph $\mathbb{G}$ are Markov chains, *i.e.*, the future state depends on the present state only and not on past states. The transition probability ρ_{ij} is computed as follows.

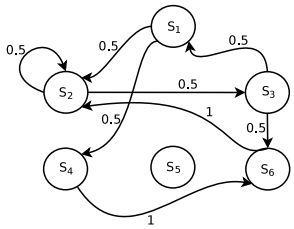
$$\rho_{ij} = \frac{\text{count}(S_i \rightarrow S_j)}{\sum_{k=1}^{284} \text{count}(S_i \rightarrow S_k)} \quad (1)$$

where, $S_i \rightarrow S_j$ represents a transition from S_i to S_j. As

discussed earlier, the paths of graph $\mathbb{G}$ are Markov chains. Therefore, the computed transition probability must satisfy Markov property [33] as given in Equation 2.

$$\sum_{i=1,j=1}^{284} \rho_{ij} = \begin{Bmatrix} 0 & \text{if all entries in } i^{th} \text{ row are zero} \\ 1 & \text{otherwise} \end{Bmatrix} \quad (2)$$

For example, consider the execution trace $\xi = \{S_1,S_2,S_3,S_1,S_4,S_6,S_2,S_2,S_3,S_6\}$ of a program $\mathscr{P}$. For $\mathbb{S} = \{S_1,S_2,\cdots,S_5,S_6\}$, Figure 1 shows the corresponding graph $\mathbb{G}$ and matrix for this example. Here, the set of distinct system-calls invoked by the program $\mathscr{P}$ is (S_1,S_2,S_3,S_4,S_6). The system-call S_5 is an isolated node as it is not invoked during execution of $\mathscr{P}$. Edges are directed and labeled with transition probability ρ_{ij}. For instance, in the execution trace ξ, two transitions $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_6$ occurred from node S_3, therefore, both the edges are labeled with equal probability, *i.e.*, 0.5. The matrix representation of $\mathscr{P}$ shows a 6×6 square matrix called transition probability matrix (TPM). Every row in TPM adds either to 1 or to 0. TPM in our case is 284×284 as $|\mathbb{S}|$ for Windows XP (SP2) is 284.



	S_1	S_2	S_3	S_4	S_5	S_6
S_1	0	0.5	0	0.5	0	0
S_2	0	0.5	0.5	0	0	0
S_3	0.5	0	0	0	0	0.5
S_4	0	0	0	0	0	1
S_5	0	0	0	0	0	0
S_6	0	1	0	0	0	0

Fig. 1: An Example of Ordered System-Call Graph (OSCG) and Transition Probability Matrix (TPM).

B. Specifying Program Behavior

The proposed approach is based on semantically-relevant paths. This concept is inherited from information theoretic model and first introduced by Cui *et al.* [19] in the domain of software testing. According to AEP, "for a random process there exists few paths that carry much more information than the other paths of the graph" [18]. The authors have proved this concept and named these paths as 'typical paths'. In literature, AEP has been applied successfully to the identically independent distributed processes and Markov chains [18]. In this paper, we also follow the same concept and hypothesize that there exist paths that are more probable than the other paths of OSCG. A path $\mathbb{P}$ of a graph $\mathbb{G}$ is defined as follows.

Definition 2 A path $\mathbb{P} = \{S_1,S_2,\cdots S_n\}$ is an alternate sequence of nodes and edges of $\mathbb{G}$ which starts from S_1 and ends at S_n.

Here, S_1 denotes S_{start} and S_n represents S_{end}. S_{start} is the first system-call invoked and S_{end} is the last system-call invoked during execution of a program $\mathscr{P}$. Each link in a path is expressed by its transition from one system-call to the other. For any path between two nodes of OSCG, path probability is computed from transition probability of its constituent links. The path probability $Pr(\mathbb{P})$ of a path $\mathbb{P}$ is given by $Pr(\mathbb{P}) = Pr(S_1).Pr(S_n = S_n|S_{n-1} = S_{n-1},\cdots,S_1 = S_1) = Pr(S_1)\cdots Pr(S_n = S_n|S_{n-1} = S_{n-1})$. The $Pr(S_1)$ is the initial probability of node S_1. The initial probability of a node S_i is the probability of occurrence of S_i among all the system-calls invoked in the execution trace ξ. Paths containing links with high transition probability are likely to contribute more to the semantic quotient.

We aimed to compute all the paths originating from S_{start} to S_{end} nodes of formed OSCG for extracting semantically-relevant paths of a program $\mathscr{P}$. Computing all-paths between two nodes is an NP-complete problem [34]. To resolve this, we approximate this phase by extracting candidate paths instead of all paths. In order to determine if a path is semantically-relevant, we apply AEP on each candidate path $\mathbb{P}$ of the sample. For this, we first determine the maximal entropy rate λ^* of the binary program under consideration as follows [19].

$$\lambda^* = max\left\{\lim_{n\rightarrow\infty} \frac{log(T_n)}{n}\right\}, \quad (3)$$

Here, T_n is the total number of paths of length n in $\mathbb{G}$. Using λ^*, we extract the semantically-relevant paths that carry nontrivial amount of information. Now, in order to define ε-semantically-relevant path with $\varepsilon > 0$, we apply following two properties (Equation 4 and Equation 5) on each path $\mathbb{P}$:

$$\left|\frac{1}{n}\log\frac{1}{\Pr(S_1,S_2,\cdots,S_n)} - \lambda^*\right| < \varepsilon, \quad (4)$$

$$\frac{\log B(S_1,S_2,\cdots,S_n)}{n-1} > \frac{1}{2}(\lambda^* - \varepsilon), \quad (5)$$

Where $B(S_1,S_2,\cdots,S_n) = \prod_{1\leq i\leq n} b(S_i)$ and $b(S_i)$ is the branching factor of the node S_i. The left hand side (LHS) of Equation (5) is average logarithmic branching factor used for constructing our feature space. We select ALBF metric of each path to construct our feature space as branching factor is a good indicator of semantic relatedness [35]. Now, we can define ε-semantically-relevant paths as follows: (Definition 3).

Definition 3 A path $\mathbb{P} = \{S_1,S_2,\cdots,S_n\}$ is ε-semantically-relevant if it satisfies *Property* 1 and *Property* 2 (Equations 4 and 5).

We use $\mathbb{T}(\varepsilon)$ to denote ε-relevant set, a set of all ε-semantically-relevant paths of the program $\mathscr{P}$. The paths in $\mathbb{T}(\varepsilon)$ vary according to the value of ε. If $\varepsilon_1 < \varepsilon_2 \cdots < \varepsilon_k$, $\mathbb{T}(\varepsilon_1) \subseteq \mathbb{T}(\varepsilon_2)\cdots \subseteq \mathbb{T}(\varepsilon_k)$. A very small value of ε may not capture all paths needed to encapsulate information content whereas a high value of ε may include irrelevant/redundant path lowering the information content of $\mathbb{T}$. We have kept the value of ε ranging from 0.5 to 7.5. The upper limit of ε is the maximum value (7.59) of *Property* 1 (LHS) in all paths of malware datasets. The initial value of ε is considered 0.5 that is greater than 0. With respect to each value of ε, we train our model with the features relevant for specifying malicious behavior. The authors in [19], have proved two theorems , which ensure that typical (semantically-relevant) paths carry relevant information of the graph. The theorems are stated as follows:

Theorem 1: Let $\varepsilon > 0$. The ε-typical paths take probability 1, asymptotically; *i.e.*,

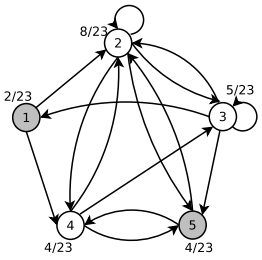
$$\limsup_{n\to\infty} \Pr\left(\left|\frac{1}{n}\log\frac{1}{\Pr(S_1,S_2,\cdots,S_n)} - \lambda^*\right| < \varepsilon\right) = 1.$$

Theorem 2: Let $\varepsilon > 0$. For any path $\mathbb{P}$ of $\mathbb{G}$ achieves λ^*,

$$\limsup_{n\to\infty} \Pr\left(\frac{logB(S_1,S_2,\cdots,S_n)}{n-1} > \frac{1}{2}(\lambda^* - \varepsilon)\right) = 1.$$

Theorems 1 and 2 have been proved by employing limsup definition of entropy rate instead of limit definition. Proving AEP with the limit definition, as in Shannon-Mcmillan-Breiman theorem [18], is difficult as it requires a strong side condition (ergodicity). In the present context, malware does not constitute same behavior averaged over time and does not exhibit ergodicity. Therefore, we can also consider the limsup definition of typical paths. Assuming the theorems and proofs are valid, we apply their concept of typical paths towards semantically-relevant paths in our approach. The set of semantically-relevant paths is not unique as a given program may have multiple execution traces due to the presence of conditional constructs (triggering of different constructs can invoke different executions). Any execution trace that results in invocation of malicious activity should suffice to extract semantically-relevant paths capturing malicious behavior. We have constructed OSCG from the single execution trace as exploring more execution traces shall add to monitoring overhead.

1) Semantically-relevant Path Extraction: Consider the Transition Probability Matrix (TPM) and graph $\mathbb{G}$ as shown in Figure 2. Here, we have a total of five nodes in the graph and the matrix that show the transition from one node to the other. In the example, S_{start} is the node 1 and S_{end} is the node 5. All possible cycle-free paths from node 1 to 5 are determined. We get a total of 9 paths. Then, we determine the value of λ^* considering all possible path lengths of 2,3 and 4. Table I shows the values as computed by application of Equation (4). These values range from 1.85 to 2.765 so we can select the values for ε in the specified range. With ε = 2.1, we get the paths $\mathbb{P}_7$ and $\mathbb{P}_8$ and for ε=2.6, we have $\mathbb{P}_2$, $\mathbb{P}_3$, $\mathbb{P}_6$, $\mathbb{P}_7$, and $\mathbb{P}_8$ as candidates for semantically-relevant paths. Selecting the different values of ε and applying Equation (5) will give us different sets of semantically-relevant paths. With these different sets of paths, we train our model and observe detection accuracy.



	1	2	3	4	5
1	0	1/2	0	1/2	0
2	0	2/8	3/8	1/8	2/8
3	1/5	2/5	1/5	0	1/5
4	0	2/4	1/4	0	1/4
5	0	1/3	0	2/3	0

Fig. 2: Ordered System-Call Graph (OSCG) and Transition Probability Matrix (TPM).

TABLE I: Paths from node 1 to 5 showing values w.r.t. Equation (4).

Paths	**Probability of the paths**	**Values of Property 1**
$\mathbb{P}_1$: 1–2–5	$\Pr(\mathbb{P}_1) : 0.0108$	2.765
$\mathbb{P}_2$: 1–2–3–5	$\Pr(\mathbb{P}_2) : 0.0032$	2.100
$\mathbb{P}_3$: 1–2–4–5	$\Pr(\mathbb{P}_3) : 0.0013$	2.530
$\mathbb{P}_4$: 1–2–4–3–5	$\Pr(\mathbb{P}_4) : 0.0002$	2.675
$\mathbb{P}_5$: 1–4–2–3–5	$\Pr(\mathbb{P}_5) : 0.0016$	2.925
$\mathbb{P}_6$: 1–4–3–5	$\Pr(\mathbb{P}_6) : 0.0021$	2.330
$\mathbb{P}_7$: 1–4–2–5	$\Pr(\mathbb{P}_7) : 0.0054$	1.850
$\mathbb{P}_8$: 1–4–3–2–5	$\Pr(\mathbb{P}_8) : 0.0010$	2.095
$\mathbb{P}_9$: 1–4–5	$\Pr(\mathbb{P}_9) : 0.0108$	2.765

IV. Verifying and Learning Malware Detection

This section presents our learning-based model that discriminates benign and malware programs. We aim to exploit semantically-relevant behavior of these programs using machine learning techniques and propose a model capable of identifying the suspicious behavior of malware binaries. The obtained high detection accuracy confirms the efficiency of the characterized behavior. In addition to that, we also show that the proposed model is evasion-resistant against system-call injection attacks.

For our proposed learning-based detection model, we construct feature space $\mathbb{F}$ by utilizing extracted semantically-relevant paths. We have used 'histogram binning' technique [36] (mainly applied in the fields of information retrieval, image processing and text processing) as it incorporates approximate matching and reduces sensitivity to slight changes in system-call sequences. This avoids the possibility of evasion encountered due to detection-aware malware. ALBF metric has been employed to determine the bin to which a semantically relevant path belongs to. In our case, each bin corresponds to a range of ALBF values. These bins are spaced at uniform intervals and hold the frequency count of respective semantically-relevant paths. These bins are considered as features. For example, if feature space consists of three bins b_1, b_2, b_3 and for program $\mathscr{P}_1$, respective bin frequency counts are f_1, f_2, f_3, its feature vector shall be $\langle f_1, f_2, f_3\rangle$.

Selecting appropriate number of bins for building our feature space involves a tradeoff between less detailed features (small number of bins imply coarser granularity and loss of information) and overly detailed features (too many bins result in loss of generalization and flexibility). We determine maximum ALBF value corresponding to malicious binaries. Dividing this by bin size yields number of bins. We constructed and evaluated feature space with bins sizes of 1, 5, 10 and 15 and observe the detection accuracy. The initial results indicate that the bins formed with a range interval of 5 (bin size is five) identify benign and malware samples more accurately. The feature vector containing bins with higher intervals merges the relevant paths of different ALBF values and may result in information loss. This merging also reduces number of elements in a feature vector. Therefore, we set the interval of 5 and consider 310 non-overlapping bins as features into our feature vector.

The constructed Feature Vector Table (FVT) is trained using learning algorithms. We use an ensemble-based learning

algorithm *i.e.*, Random Forest, [37], [38] for differentiating malware and benign samples. Since, it provides a better generalization of information even in the presence of noise and therefore widely used in the literature. It is a collection of many decision trees. It is primarily used when the data set is very large. The decision tree is constructed by randomly selecting subsets of the training data and attributes that can partition the available data set. The final decision tree is an ensemble (*i.e.*, a collection) of forest such that each decision tree contributes towards the classification of instances.

V. Experimental Setup and Results

The experiments are performed on Intel Core i3 2.40 GHz with 2.8 GiB RAM, running on Ubuntu 12.04 operating system. For capturing the system-call traces we use Xen hypervisor [39] and create a virtual environment using Ether. The underlying guest OS in Ether is Windows XP (SP2). Therefore, we have built our prototype model by executing binaries in Windows XP. Although, Microsoft abandoned its support for XP but still it is a popular OS widely used in various government agencies, banks and in ATMs. As a result, existing recent similar approaches [6], [11], [14], [40] also utilize XP. However, the proposed approach is not specific to particular OS and analysis framework as: 1) the target malicious binaries (PE format) affect all Windows platforms, and 2) system-call sequence used in Windows XP is a subset of those utilized in Windows 7 [32]. It will perform in a similar fashion if system-call traces are collected with different Windows OS and some other analysis framework (Anubis, Cuckoo, GFISandbox, to name a few). In this section, we present the implementation details and evaluation of our proposed approach. The experiments are carried out using benign and malware executable samples. The proposed approach detects the malicious Windows PE binaries (PE is the most popular file format among malware authors as reported by the *virustotal.com* [41]).

A. Experimental Dataset

We utilize real instances of malware and benign samples. The majority of malware detection approaches [40], [42] make use of one malware dataset to evaluate the performance of their approach. These approaches perform well on selected dataset, however, do not generalize well to other datasets and result in performance degradation. Therefore to evaluate generalization of our approach, we used two different malware datasets and label them as D_{old} (old dataset) and D_{new} (new dataset) . The former dataset consists of 1209 samples. This set also includes samples utilized in [43] for their work of detecting metamorphic malware. The types of samples in this set include packed, polymorphic and metamorphic malware. We selected this dataset for two reasons: 1) to represent the class of malware samples discovered prior to 2012, and 2) to estimate the performance of our method with morphed and packed samples. The latter set (D_{new}) of samples is downloaded from the malware repository system, *i.e.*, *VirusShare.com*. The mentioned repository system labels each uploaded sample after scanning it with 55 AV scanners. We can rely on the labeling process of *VirusShare.com* as it is akin to comparing with large number of AV scanners. This dataset consists of 1226 malware samples each of which was discovered from January 2013 to March 2014 and it is labeled as 'new'. Both datasets are divided into training (70%) and test (30%) set.

We used one benign dataset that contained total number of 1316 samples. Benign samples are scanned by uploading them to the web portal *VirusTotal.com* to verify their non-maliciousness. Our benign dataset consists of different kinds of software applications such as browsers, games, filezilla, googletalk setup, iTunes, youtubedownloader, Media players, wireshark, to name a few. We used these benign software programs to evaluate the accuracy of proposed model.

As discussed earlier, we monitored the execution of each benign and malware sample in the controlled environment created using Ether. We observed that during execution there were some malware samples that did not generate any log and few benign and malware samples that cause executional errors. The malware samples embedded with anti-detection features (VM-aware and trigger-based) do not generate any log. The execution errors occurred due to OS compatibility issues. Our final datasets (benign and malware) include the samples that are executed in guest OS without abnormal termination, without execution errors and without generating any logs.

To generate system-call logs, each sample is permitted to execute for 10 minutes. According to [44], five minutes is sufficient duration for the execution monitoring. We doubled this execution time to capture the malware equipped with capability of carrying out time-out attacks. We observed that in all samples, benign as well as malicious, the execution sequences are mostly made of 160 different calls out of 284 calls. We refer these calls as 'frequent' calls. Remaining 124 calls are regarded to as 'rare' in following discussions. The experiments are performed with training sets of benign and malware datasets, and the performance evaluation is carried out using test samples.

B. Approximate All Path Computation

We constructed OSCG for each sample as discussed earlier to determine the paths between S_{start} and S_{end}. Identifying all paths between two nodes of a graph is an NP-complete problem [34]. The time complexity of computing all paths is exponential in case of a complete graph. To reduce the time complexity, we approximate the 'all path computation phase' of our approach. We, first investigate if an OSCG is sparse. We observe the average link-count in OSCGs of benign and malware samples of our datasets. The average link-count of benign samples is 174 and that of malware samples is 282 indicative of the sparse nature of our OSCGs as the number of edges is in $\mathcal{O}(|\mathbb{S}|)$, where $\mathbb{S}$ is the number of vertices (284) in the graph.

To approximate the all-paths phase, we conducted an experiment with 500 malware and 500 benign samples. The samples are selected in a manner that they cover the entire range of link-counts. With these samples, we exhaustively computed all paths from S_{start} and S_{end} and computed average number of

paths for a given path-length. Figure 3 shows this distribution for both benign and malware samples. As can be seen here, average number of paths is normally distributed for benign as well as malicious files. However, in path-length ranging from 10 to 31, the average number of paths in malware samples is more than the benign samples. Paths in this range can be used for discriminating a malware from benign. We have used the paths in this particular range as candidate paths for extracting semantically-relevant paths. Instead of computing all paths for all the samples we computed the candidate paths (approximation of all paths), which reduces the path computation time for all the samples.

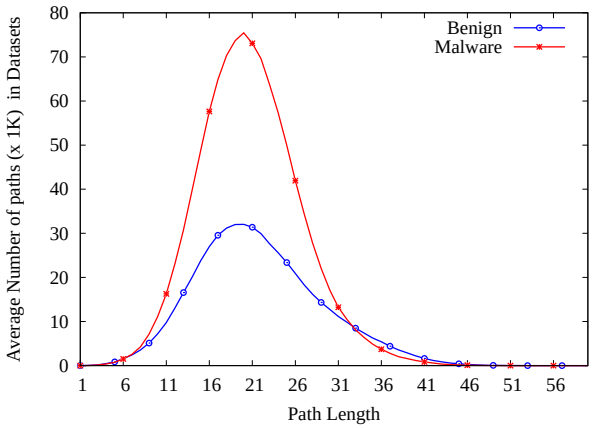


Fig. 3: Path distribution w.r.t. lengths in benign and malware datasets.

Table II shows the average time consumed in determining all paths and candidate paths. We observe that time taken in computing all paths is higher than the candidate paths. It means that using candidate path we can reduce the overall computation time. The maximum time consumed was ~ 11 hours and ~ 9.7 hours for all-path computation and candidate-path computation respectively. We also observe that there are some samples, which required higher processing time than the samples having higher link-count. This indicates that processing time is not directly proportional to link-count. The processing time for this particular phase needs to improve for practical application. In the majority of samples ($\sim 88.3\%$), candidate paths have a link-count of atmost 450 which result into a total time of ~ 1.83 hours.

TABLE II: Processing time of all-paths and candidate-paths.

Link-count up to	**Avg. Time of All-paths (in sec.)**	**Avg. Time of candidate-paths (in sec.)**	**% of Samples Covered**
50	0.01	0.005	0.98
51-150	1.78	0.47	6.07
151-250	450.64	162.89	37.89
251-350	9271.65	1145.72	27.32
351-450	19080.20	5292.86	16.08
451-550	24848.32	15565.60	9.84
551-650	33482.56	28384.77	1.33
651-750	39524.45	35109.9	0.49

We determined the candidate-paths for all the samples and, then, extracted the semantically-relevant paths as explained in Section III. We constructed $\mathbb{T}(\varepsilon)$, ε-relevant set of benign and malware samples and feature vectors for all binaries using the frequency distribution of ALBF values of their $\mathbb{T}(\varepsilon)$. The constructed feature vector is trained using Random Forest as described earlier. The above process has been repeated for different ε values.

C. Detection Accuracy

We have evaluate the performance of our proposal in terms of popular evaluation metrics [45], [46] – *i.e*, True Positive Rate (TPR), False Positive Rate (FPR), True Negative Rate (TNR) and False Negative Rate (FNR). In the present context, we designate malware class as positive and benign class as negative. TPR (FPR) is the fraction of malware instances correctly (incorrectly) classified. Similarly, TNR (FNR) denotes the fraction of benign instances correctly (incorrectly) classified. For any malware detection model, it is desired that TPR should be high and FPR and FNR should be low. For two datasets D_{old}, D_{new} considered in our evaluation, Table III summarizes TPR and TNR for different values of ε.

TABLE III: Detection accuracy of D_{old} and D_{new}.

ε	D_{old}		D_{new}		**Overall**
	TPR	**TNR**	**TPR**	**TNR**	**Acc**
0.5	0.3	100.0	1.7	100.0	50.50
1.0	4.5	32.7	18.1	99.2	38.62
1.5	64.0	71.3	44.6	62.7	60.65
2.0	81.8	88.4	77.9	78.1	81.55
2.3	92.3	91.5	88.4	93.3	91.37
2.6	96.2	95.3	94.6	93.8	94.97
2.9	**95.9**	**96.1**	**94.3**	**95.4**	**95.42**
3.2	90.3	94.3	94.7	96.1	93.85
3.5	91.4	95.1	92.3	93.9	93.17
4.0	88.6	92.6	89.1	91.5	90.45
4.5	87.9	88.9	90.3	89.0	89.02
5.5	87.1	89.2	89.3	89.6	88.80
6.5	84.1	88.6	87.1	90.3	87.52
7.5	84.6	87.1	85.0	87.2	85.97

As can be seen from Table III, $\varepsilon \in \{2.3, 2.6, 2.9, 3.2\}$ yields higher detection accuracy. It can be easily deduced from the table statistics that our constructed feature space has the ability to discriminate between malware and benign samples. Our model achieves the highest accuracy of 95.425% at $\varepsilon = 2.9$. Therefore, we have selected it as a threshold. We conducted this experiment extensively with various values of ε to validate our hypothesis that lower values of ε excludes some of information-rich paths. This is reflected in poor performance exhibited by initial rows of Table III. Too many paths, as happens at higher values of ε, can lead to generalization and, therefore, result into the decrease in detection accuracy.

The misclassified instances are shown in Table IV. As can be seen, our model performs best at selected threshold of ε=2.9. For malware classification, FPR and FNR should be low as a high value of FPR shall result in malware being considered benign. A high FNR may prohibit execution of legitimate applications. With D_{old} samples, we achieved 3.9% and 4.6% of FPR and FNR respectively. Similarly, FPR of 4.1% and FNR of 5.7% is obtained with D_{new}.

Some of the malware samples yield only partial logs, and this has contributed towards FPR. During runtime, these

samples terminated very quickly and did not reveal their actual payload. In our case, this behavior was observed with samples belonging to `worm.autorun` malware family. The samples of this family try to infect the system by creating `.inf` file on root directory of system. When these files detect the presence of virtual environment, they do not reveal their malicious payload and terminate the execution. Hence, these instances were misclassified. A false negative is observed when a legitimate monitored application shows high similarity with the malicious samples. We found that the system-calls related to memory access, process and thread handling activities were common to malware samples. Any benign application using these calls may show high correlation with malware samples and may be misclassified. This aids to FNR.

TABLE IV: False rate with D_{old} and D_{new}.

ε	D_{old}		D_{new}	
	FNR	**FPR**	**FNR**	**FPR**
0.5	99.7	0	98.3	0
1	95.5	67.3	81.9	0.8
1.5	36	28.7	55.4	37.3
2.0	18.2	11.6	22.1	21.9
2.3	7.7	8.5	11.6	6.7
2.6	3.8	4.7	5.4	6.2
2.9	**4.1**	**3.9**	**5.7**	**4.6**
3.2	9.7	5.7	5.3	3.9
3.5	8.6	4.9	7.7	6.1
4	11.4	7.4	10.9	8.5
4.5	12.1	11.1	9.7	11
5.5	12.9	10.8	10.7	10.4
6.5	15.9	11.4	12.9	9.7
7.5	15.4	12.9	15	12.8

To reduce the false alarm rate in our approach, we need to identify and remove the call-transitions that are common to OSCGs of most of the malware and benign samples. For this, we may first extract the common subgraph (using graph isomorphism [47]) from all malware and benign samples and then the edges of this subgraph can be removed from all the OSCGs. However, the false alarm rate in our approach is considerably low when compared to approaches [6], [40], and [20] in which the false alarm rate of 10.9%, 9.8% and 9.7% is observed respectively.

The detection capability of our approach with unknown samples (test samples that are not used in training phase) is evaluated using two test datasets with both the training models prepared with ε value as 2.9. We performed testing of our both the test sets. For the first test set, we observed overall detection accuracy of 94.2% (D_{old}: 94.8%, D_{new}: 94.7%). In case of second test set, we achieved an overall accuracy of 93.4% (D_{old}: 93.7%, D_{new}: 93.1%). The detection accuracy of test samples is approximately similar to that of our trained model. There is a minor difference in detection accuracy of both the sets and this was expected because the learning-based models always perform better with training samples due to the implicit knowledge about the samples. Similar trends of false detection rate are observed with test samples. Our experimental results indicate that the proposed method is effective in discriminating the benign and malware instances.

D. Resilient against dynamic obfuscation

As discussed earlier, modern malware inserts irrelevant and independent system-calls to evade the system-call based detection approaches that rely on either signature or exact pattern matching. These solutions are evaded by malware authors as these methods directly work on raw system-features such as opcodes, instructions, hexbytes, and *etc.* that can be obfuscated or replaced by equivalent alternate features. The discriminating components are clearly visible and hence tampered by malware writers. On the other hand, our method provides a solution by employing a feature space that is not linearly related to raw system features and hence opaque to malware writers.

To measure the robustness of proposed approach in the presence of system-call injection attack, we performed experiments on two sets of system-calls, *i.e.*, rarely invoked system-calls (*RISC*) and frequently invoked system-calls (*FISC*). The former set consists of calls that are rarely invoked by malware or benign applications in our datasets. As stated earlier, the malware and benign applications mostly utilize 160 calls out of 284. Therefore, remaining calls serve as irrelevant to both the applications and therefore become valid candidates of injection attacks. The latter set includes frequent calls invoked by benign and malware applications. For this, we have selected the benign sample from the set containing 100 benign programs, which are considered on the basis of larger trace size among all the other benign samples. We also evaluated the impact of considering single or different benign programs on our approach. For single program we selected the sample `AdbRdr930_en_US.exe` containing more than 10^5 calls. We observed that in both cases the results are nearly the same. It indicates that considering different benign programs for injection does not affect our approach.

In both the experiments, we inserted system-calls into random locations of the execution trace of randomly selected malware programs. The malware samples considered for this are the test samples of D_{new} dataset as these samples belong to the class of latest malware attacks. The number of calls in these malware traces ranges from 674 to 124652. We inserted total calls that are 10%, 20%, $\cdots$, 100%, 150%, 200% of malware traces. For inserting system-calls, we adopt the strategy followed by authors in [11]. These calls are injected one at a time and at random positions in the malware traces. For both the experiments, we observed the performance of our model with ε value of 2.9 and results are shown in Figure 4.

Figure 4 illustrates the performance of our model with both the experiments in terms of detection accuracy. The detection accuracy of our model does not vary up to 30% of call-injection rate. For some malware samples, this translates to injection of $\sim$30000 calls. We exhaustively injected calls into malware traces and in this way injection also occur in extracted semantically-relevant paths and therefore beyond 30% call injection rate we observed the fall in detection accuracy.

Figure 5 shows TPR and FPR values with respect to both the experiments. It means that the discriminating patterns of proposed approach are not affected by the injected calls. However, when we increase the injection rate, the detection

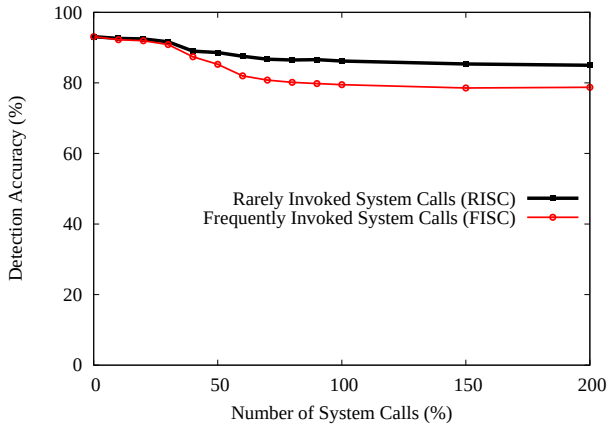


Fig. 4: Detection capability in presence of behavior obfuscation with *RISC* and *FISC*.

accuracy starts decreasing. This fall in the detection accuracy is expected as when we insert more calls into the malware traces, TPM no longer matches the modified samples as more paths are added into semantically-relevant set affecting its frequency distribution. This is expected as insertion of rarely invoked system-calls does not affect TPM as it is akin to adding some transition to almost isolated nodes and such paths are unlikely to be included in semantically-relevant set unless a large number of injection takes place. With *RISC*, our model performs better when compared to *FISC*. By inserting benign call sequences, we observe an increase in false detection rate of our model. The maximum decrease of 6.9% and 13.6% in the detection accuracy is observed for RISC and FISC experiments respectively. The feasibility of inserting calls of RISC set is more than the FISC as the latter set of calls can affect the prime objective of malware. Therefore, our method shall work without much loss in detection accuracy.

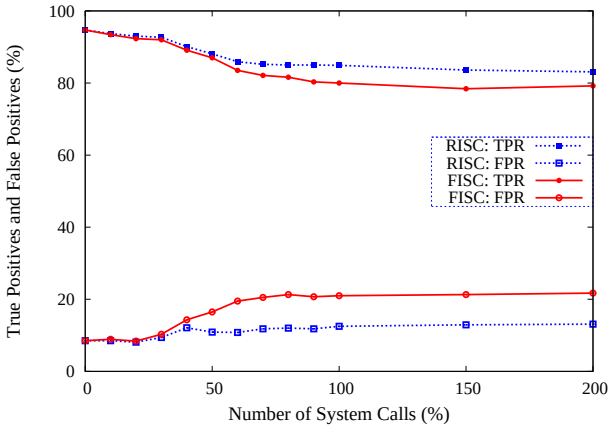


Fig. 5: True Positives and False Positives with *RISC* and *FISC*.

The other important concern of call-injection in our approach is to modify the S_{start} and S_{end}. For this, we closely inspect the variation in ALBF value of paths after adding two irrelevant calls, *i.e.*, S'_{start} and S'_{end} at initial and last position of malware trace. By doing so, we observed that the semantically-relevant set contains same paths with two additional links, *i.e.*, $S'_{start} \rightarrow S_{start}$ and $S_{end} \rightarrow S'_{end}$ showing single outgoing transition. The ALBF metric (Equation 4) is sensitive to path-length as well as the branching factor. If two additional links are added that were not there in the previous OSCG, then only path-length is affected and increased by 2. The branching factor remains same as the link-count is 1 for both the links. A negligible fall in ALBF is observed due to increase in the path-length. This fall in the majority of cases does not change the bins of those modified paths hence it will not affect our approach. Now, if a long sequence of unrelated calls is added (pre/post) to just increase the path-length then in very few cases it will affect the ALBF as we have restricted the path-lengths (from 10 to 31). To evade the approach, malware authors have to append and prepend a long sequence of unrelated calls with higher outgoing transitions, which modify the bins of all the paths in such a way that increases the false alarm rate.

E. Comparison with existing approaches

In the previous section, we have shown that our approach has resulted into better detection accuracy. However, we need to compare with other approaches to asses the quality of our results. Here, we present a comparative evaluation with current state-of-art dynamic malware detection techniques. Moreover, we analyze the impact of call-injection attack on our approach to one proposed by Park *et al.* [11].

Compared to previous work, our proposed approach shows improved malware detection rates. Figure 6 shows the TPR

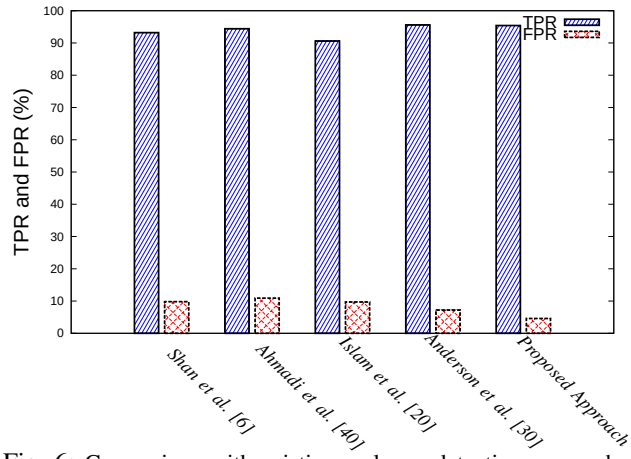


Fig. 6: Comparison with existing malware detection approaches.

and FPR of every approach. From this figure, our proposed approach is shown to outperform other methods, with the highest true positives and the lowest false positives. The better performance of our approach is due to semantically-relevant paths, which represent the program semantics that cover the most relevant behavior of malware and benign programs.

Park *et al.* [11] proposed a graph clustering method [48] for deriving the common behavior of malware samples. The authors performed an abstraction from system-call traces and used kernel objects [49] to represent malware behavior. Further, they applied graph matching and determined a threshold to asses the detection rate of their approach. Moreover, the

authors have built a kernel object behavior graph (KOBG) to exploit the dependency between the kernel objects. The kernel objects and their dependencies information is extracted from the system-call traces acquired using Ether framework. To evaluate Park's approach on our samples, we adapted their approach as mentioned in [11]. We constructed KOBG in similar fashion and built a weighted common behavior graph (WCBG) using McGregor algorithm. To implement the algorithm, we made use of graph C++ Library provided by the Boost Software [50].

Figure 7 contrasts the performance decay of our proposed with the one in [11] for call-injection rate ranging from 0% to 100%. It is quite clear that our approach outperforms the approach in [11]. In our approach, the maximum fall observed is ∼13% while in [11] the observed maximum fall is ∼23%. The detection accuracy of Park's approach with 0% injection rate is observed as 92.45%. The false alarm rates of their approach are 8.2 (FNR) and 6.9 (FPR). The approach by Park *et al.* [11] is based on exact-pattern matching as a result of which the call-injection attack and false alarm rates result into higher performance deterioration. However, our approach is not based on exact pattern matching, but abstracts semantically-relevant paths as bins of branching factor. Therefore, it is less vulnerable to call-injection attack.

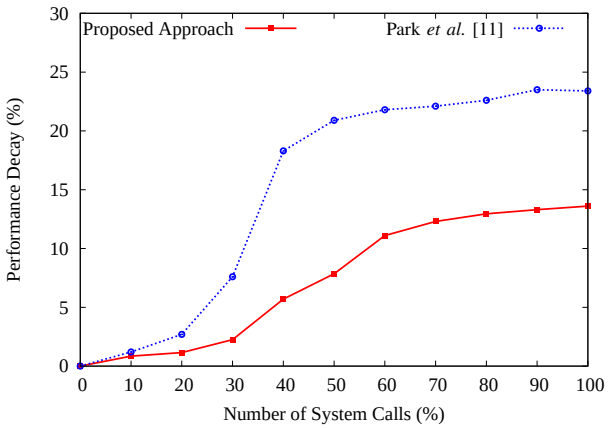


Fig. 7: Comparison of proposed approach and approach in [11].

VI. Discussion

In this paper, we introduced the concept of specifying program semantics in order to discriminate the malicious from non-malicious binaries. To address this, we abstracted the system-calls to a higher level and created sets containing semantically-relevant paths. These semantically-relevant paths cumulatively represent the program semantics since each path sequence exhibits a specific functionality of the program. In this section, we discuss the merits and demerits of proposed approach. For any malware detection approach, it has to address the issues such as generality, resiliency, stealthiness and associated overhead.

A. Generality

Generality determines the ability of detection model to scale uniformity with 1) comprehensive set of malware samples, and 2) known (training) and unknown (test) malware instances. We used two different datasets that include a wide spectrum of malware samples. The overall detection accuracy with both the datasets D_{old} and D_{new} are observed as 96% and 94.85% respectively. Both the figures are nearly the same with a marginal difference of 1.15% occurred due to the presence of malware samples in D_{new}, which do not manifest their malicious behavior during runtime. These samples are termed as detection-aware malware. The detection-aware malware senses the presence of instrumented virtual environment. In the proposed approach, to create the virtual environment, we used Ether. Ether can be detected as the BIOS data strings for Ether make use of emulated variant from Bochs virtual machine. Moreover, the Ethernet card that is emulated by underlying Xen system can be analyzed easily. The detection-aware malware exploits these variations and ensures that it cannot be analyzed. As a result, it generates partial log or no log during runtime. Our datasets do not include the samples with no logs. Therefore, the only concern is the generated partial logs that result into misclassification. Although, in our case the false alarm rates were significantly low as compared to other existing approaches [6], [20], [30], [40]. This particular limitation is common to the majority of dynamic malware detection approaches. In future, we can substitute Ether with more resilient framework or we may augment more than one framework (emulated, virtualized, and instrumented) to retrieve complete logs of detection-aware malware.

The other factor that assures the generality is the performance of our detection model with known and unknown malware samples. We observed the uniformity in our training (95.42%) and testing (93.8%) results. In our case, the difference in training and testing results is only 1.6% which is negligible. Hence, our model achieves the generality and thus capable of detecting a wide range of malware instances.

B. Resiliency

Resiliency refers to the robustness of the proposed approach in the presence of possible evasion embedded into malware files. As our proposed model relies on system-call traces, one possible evasion technique to thwart our model is system-call injection attack. Using this attack, malware authors modify the system-call sequences of malware binaries at run time. For incorporating this, the malware authors either make modifications into the malware program or create new binaries through the injection of system-calls. We ran two different experiments to evaluate the robustness of our approach against behavior obfuscation. The experimental results indicate that the detection accuracy remain invariant up to 30% of call injection rate. Beyond this, we observed a fall in detection accuracy that stabilizes above an injection rate of 70%. The system-call sequences of our malware dataset contain on an average more than 10^5 calls. Inserting even 10K, 20K and 30K independent calls into malware traces does not affect the proposed mechanism. Furthermore, our dataset consists of packed, polymorphic and metamorphic samples which indicate that the proposed approach can complement existing static malware detection methods.

C. Stealthiness

Stealthiness refers to the detection capability by which our approach operates with high detection accuracy without disclosing discriminating patterns to malware attackers. The discriminating component of our approach is neither a sequence of system-calls nor a feature space linearly derived from these sequences. The discriminating component of our method is composed of the ranges of ALBF values. As these values are accumulated in bin, our feature space is non-linearly related to the sequence of calls. Multiple semantically-relevant paths imply different subsequence of calls being used in the construction of feature space. Modification in one path shall not impact performance of the proposed model. Only large modifications in transitions of all semantically-relevant paths will affect our model. The modification is complex as the attacker needs to identify all semantically-relevant paths and modifying the path sequences in a way that it substantially modifies ALBF bins. Hence, our proposed approach provides stealthiness and is resilient against present and future malicious threats.

D. System Overhead

In conjunction to its detection accuracy and resiliency against call-injection attacks, we also discuss the associated overheads of the proposed approach. Table V shows the best, average and worst time per sample during the main steps of our approach. The total time shown in the Table V does not include monitoring time as it is common to all behavior-based approaches. Each step is discussed as follows.

TABLE V: Best, Average and Worst Processing time of each sample

Main Steps	**Best Time (in sec.)**	**Average Time (in sec.)**	**Worst Time (in sec.)**
System-Call Monitoring	–	–	600
OSCG Construction	0.001	0.02	1.2
Candidate-Paths Computation	0.005	3355.86	35109.9
Semantically-relevant Paths Extraction	0.07	0.18	0.67
Training Time	1.54	1.54	1.54
Total Time -Monitoring Time	1.616	3357.6	35113.31

1) System-call Monitoring: Execution tracing of benign and malware binaries in our approach depends on Ether. Therefore, the overheads associated with Ether are inherited into our approach. We fixed the time-out of 10 minutes (600 seconds). Therefore, we observe this overhead of collecting system-call traces. Monitoring executables from Ether is a time-consuming task. Ether uses exceptions whenever a running application makes a system-call to access system services. These exceptions result into significant performance overhead. To reduce this overhead, we can use a faster analysis framework. The proposed approach is not specific to a given monitoring environment and can be generalized by applying the same methodology with other operating systems as well as virtual/sandboxing environments. To investigate this, we conducted a small experiment using 20 malware samples of D_{new} dataset. We collected execution traces of these samples from Cuckoo sandbox with 10 minutes of timeout. To extract the run-time traces of executables, we submit the sample via `submit.py`. The inline "Cuckoo Agent" (`agent.py`) receives the executable and analyzes it. After the analysis is over, a behavior report is generated which includes logs containing various parameters such as API, system-calls, static attributes, DLL invoked, PE header, and processes created during runtime. We used the generated behavior report and extracted the system-calls that are invoked. We then created the feature space for these samples in a similar fashion as mentioned earlier and closely observed the variation in frequency distribution of Ether generated traces with Cuckoo generated traces. We found out that there is a negligible variance in the frequency distribution in both the cases.

2) OSCG Construction: In this phase, we extract the system-call sequences from the acquired traces and then build the TPM using the transitions of system-calls. The processing time for TPM construction is negligible (average: 0.02 sec. and worst: 1.2 sec.) as it depends on the trace length. For samples with larger trace-length, OSCG is constructed in few seconds and for average trace length, it is constructed in few milliseconds. We can say that this phase does not lead to higher processing time.

3) Candidate-path Computation: In our approach, we determined the paths between S_{start} to S_{end}. In this quest, we observed that the time complexity for determining all paths is very high. We have shown that our OSCGs are sparse in nature as the average link-count of our samples is less than the total number of nodes. Therefore, our approach does not result into exponential time complexity. However, the processing time for computing all-paths is significantly high that it affects the applicability of our approach in real-time situations. To reduce this processing, we approximated all-path computation and determined the candidate paths by restricting the path-length. The average processing time for computing candidate-paths is $\sim$3355 seconds and the worst processing time is $\sim$35109 seconds. Though, this approximation improves the processing time of our approach yet when compared to other existing approaches it is slightly high. In order to minimize the time complexity of this phase, we can use more efficient path-computation algorithm. The authors in [34], proposed a survey of various parallel graph algorithms using systolic arrays, associative processors, array processors, and multiple CPU computers. General-Purpose Computing on Graphics Processing Units (GPGPU) provides a powerful platform to implement the data-intensive algorithm. Kaczmarski *et al.* [51] proposed an approach for accelerating the Breadth First Search (BFS) algorithm with CUDA implementation on GPU. The authors have shown the significant improvement over CPU-based implementation of BFS. The results we present show great promise in using semantically-relevant paths to classify malware, the computational complexity would be prohibitive in a real-time setting. In future, we will also create the parallel version of our path-computation algorithm and reduce the incurred overhead.

4) Training Time: Training time includes the time taken to train our feature vector. This is measured with respect to entire feature vector. It is one-time cost of the order $\sim$1.54 seconds. It is not measured per sample. Although, to keep our

system up-to-date, we need to train our model with the newer samples within fixed time interval (monthly or quarterly).

VII. Related Work

Our proposed approach addresses the problem of malware detection. The desirable property of any malware detector is that it must be capable of detecting zero-day and unseen malicious attacks. Modern malware is analysis-aware, *i.e.*, it tries to evade the existing detection mechanisms whether static or dynamic. As discussed earlier, we prefer execution-based dynamic approach over static to overcome the limitations of the latter. In this section, we review dynamic approaches of malware detection published in the literature that represent the extracted dynamic attributes in one of the following forms: 1) n-grams, 2) feature-statistics and 3) graphs.

A. n-gram based Approaches

The n-gram based representation has been deployed in many dynamic malware detection approaches [45], [52], [53]. n-gram is a contiguous sequence of n features extracted in a sliding window (moves one feature at a time) manner from program traces. The authors in [45], [52] have presented an approach that identifies the malicious behavior using Application Programming Interface (APIs) and system-calls respectively. They have applied a fingerprinting approach matching the n-gram features prominently present in malware but absent in benign applications. Yongzheng *et al.* [53] have presented a visualization approach using DotPlots to cluster similar malware instances. For this, authors have made use of byte opcodes and constructed the n-gram features. They have also applied hash-based content sampling to reduce their feature space. The n-gram based representation methods suffer from the drawback of dimensionality and depend on two parameters n (consecutive number of features to be considered) and L (total number of n-gram) [30].

B. Feature-statistics based Approaches

In literature, security researchers have utilized the statistics of extracted features to identify malicious binaries. The statistics include feature count, probability, data-value, entropy, information-gain, temporal values, to name a few. Islam *et al.* [20] have proposed an approach that uses static and dynamic attributes to classify benign and malware samples. They have applied four classifiers, Support Vector Machines, Decision Trees, Random Forest, and Instance-based to carry out their objective. A similar approach was proposed by Ahmadi *et al.* [40]. The authors have used API calls to construct their feature vector. Then, they applied two feature selection methods (fisher score and CFSSUBSETEVAL) to remove the redundant and irrelevant features. Authors in [54] used temporal values of system object and visualized the malware sample using treemaps and threaded graphs. Directly applying feature-statistics of raw feature leads to high false alarm rates as these statistics provide a low signal-to-noise ratio.

C. Graph-based Approaches

The Graph-based representation is deployed to encode the relative information of dynamic attributes. To detect malware using graph-based modeling, a graph or subgraph with aggregated feature attributes is formed. This modeling represents the dependency structure of binaries in terms of control-flow, data-flow, and information-flow. The proposed approach makes use of graph-based representation as other two have few limitations that restrict them in specifying actual malicious behavior. Our proposed approach extracts the program semantics and uses it to train our detection model. Existing approaches [8], [9], [11], [30], [55]–[57] use graph-based representation to capture malicious programs. The authors in [55], [56] have presented a visualization approach to cluster the samples showing malign behavior. Another clustering approach proposed by Jacob *et al.* [9] which specifically identifies bot-initiated Command & Control (C&C) communication. They have constructed a behavior graph of system call traces. C&C templates are created with known and unknown C&C communications. The authors claimed that these templates share similarity in behavior graph and thus samples with homogeneous C&C activities are grouped into the same clusters. A similar behavior-based malware detection approach was proposed by Shan *et al.* [6]. In their approach, authors have constructed behavior templates containing OS-level information-flow. The behavior templates represent the group of atomic behaviors. An unknown sample with suspicious behavior is marked if its behavior template is matched with one of the templates stored in database.

Shun *et al.* [57] have developed a dependency structure matrix (DSM) to show the task/module dependencies to detect the module-based co-working malware. The approach proposed by Anderson *et al.* [30] detects malware by utilizing Markov chain based dependency graphs of program traces. They have incorporated multiple kernel learning to construct a weighted combination of combined feature set (opcodes, basic blocks and system calls). Fredrikson *et al.* [8] have developed a tool named as HOLMES. It works in two steps: 1) Extraction of significant malicious behaviors and 2) Creation of a discriminative specification of malware behavior. For this, a dependency graph is constructed in which graph vertices (system-calls) are connected by dependency in their arguments. They have extracted a common synthesized malware behavior by applying structural leap mining. A similar approach has been proposed by authors in [11]. They have constructed a kernel object behavior graph (KOBG) to derive the common behavior of malware families. The authors have created a HOTPath and applied an exact pattern matching approach. The majority of these approaches make use of graph matching that is ambiguous due to graph isomorphism. On the other hand, our approach that is independent of any graph-matching and thus more robust with system-call injection attacks as compared to other existing approaches as shown by our experiments.

VIII. Conclusion

Malware detection is the first line of defense against malicious threats. Modern malware is detection-aware therefore

detecting all types of malware is a daunting task. In this paper, we proposed a new mechanism for identifying current malicious binaries which are resilient to static and dynamic obfuscation techniques. To carry out this objective, we captured the execution flow of malicious binaries in terms of system-calls and transformed them into Ordered System-Call Graph (OSCG). Then, we applied the concept of Asymptotic Equipartition Property (AEP) inherited from information theory. Using AEP, we produced a set of semantically-relevant paths from each OSCG. These paths cumulatively describe the average behavior of a binary. Our experimental results demonstrate that semantically-relevant paths can be used to infer the malicious behavior and to detect numerous new and unseen malware samples. The proposed approach shows its robustness against system-call injection attacks. In addition, we also compared our method with existing solutions. We observed that our approach was more efficient in terms of malware detection rate. Our future work will focus on the development of path computation algorithms to reduce the overhead.

REFERENCES

[1] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proc. of the 22Nd USENIX Conference on Security (SEC'13)*, 2013, pp. 637–652.

[2] AV-test. (2014, Nov.) Av–test malware statistics. [Online]. Available: http://www.av-test.org/en/statistics/malware/

[3] W. Wang, I. Murynets, J. Bickford, C. V. Wart, and G. Xu, "What you see predicts what you get–lightweight agent–based malware detection," *Secur. and Commun. Netw.*, vol. 6, no. 1, pp. 33–48, 2013.

[4] D. Spinellis, "Reliable identification of bounded-length viruses is NP-complete," *IEEE Trans. Inf. Theory*, vol. 49, no. 1, pp. 280–284, Jan. 2003.

[5] H. Qiu and F. Osorio, "Static malware detection with segmented sandboxing," in *Proc. of IEEE 8th International Conference on Malicious and Unwanted Software (MALWARE'13)*, Oct 2013, pp. 132–141.

[6] Z. Shan and X. Wang, "Growing grapes in your computer to defend against malware," *IEEE Trans. on Inf. Forensics and Secur.*, vol. 9, no. 2, pp. 196–207, Feb 2014.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. of IEEE Symposium on Security and Privacy (SP'10)*, 2005, pp. 32–46.

[8] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proc. of IEEE Symposium on Security and Privacy (SP'10)*, 2010, pp. 45–60.

[9] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "Jackstraws: Picking command and control connections from bot traffic," in *Proc. of the 20th USENIX Conference on Security (SEC'11)*, 2011, pp. 29–48.

[10] H. Lu, X. Wang, B. Zhao, F. Wang, and J. Su, "Endmal: An anti-obfuscation and collaborative malware detection system using syscall sequences," *Math. and Comput. Model.*, vol. 58, no. 5–6, pp. 1140–1154, 2013.

[11] Y. Park, D. S. Reeves, and M. Stamp, "Deriving common malware behavior through graph clustering," *J. Comput. Secur.*, vol. 39, Part B, pp. 419 – 430, 2013.

[12] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[13] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: automatically evading system-call-behavior based malware detection," *J. Comput. Virol.*, vol. 8, no. 1-2, pp. 1–13, 2012.

[14] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla, "Bee master: Detecting host-based code injection attacks," in *Proc. of 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'14)*, 2014, vol. 8550, pp. 235–254.

[15] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003, pp. 272–280.

[16] Y. Ji, Y. He, D. Zhu, Q. Li, and D. Guo, "A mulitiprocess mechanism of evading behavior-based bot detection approaches," in *Information Security Practice and Experience*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8434, pp. 75–89.

[17] M. Ramilli, M. Bishop, and S. Sun, "Multiprocess Malware," in *Proc. of 6th International Conference on Malicious and Unwanted Software (MALWARE'11)*, Oct 2011, pp. 8–13.

[18] T. M. Cover and J. A. Thomas, *Elements of Information Theory 2nd Edition (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, July 2006.

[19] C. Cui, Z. Dang, and T. R. Fischer, "Typical paths of a graph," *J. Fundam. Inf.*, vol. 110, no. 1-4, pp. 95–109, Jan. 2011.

[20] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *J. of Netw. and Comput. Appl.*, vol. 36, no. 2, pp. 646 – 656, 2013.

[21] N. Kheir, "Behavioral classification and detection of malware through HTTP user agent anomalies," *J. of Inf. Secur. and Appl.*, vol. 18, no. 1, pp. 2 – 13, 2013.

[22] R. Moskovitch, Y. Elovici, and L. Rokach, "Detection of unknown computer worms based on behavioral classification of the host," *J. Computational Stat. and Data Anal.*, vol. 52, no. 9, pp. 4544 – 4566, 2008.

[23] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware," in *Proc. of International Conference on Network and Distributed System Security Symposium (NDSS'10)*, 2010, pp. 1–16.

[24] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Proc. of Recent Advances in Intrusion Detection (RAID'11)*, vol. 6961, 2011, pp. 338–357.

[25] S. Naval, V. Laxmi, M. Gaur, S. Raja, M. Rajarajan, and M. Conti, "Environment-reactive malware behavior: Detection and categorization," in *Proc. of 7th International Workshop on Autonomous and Spontaneous Security (SETOP'14)*, 2014, pp. 1–16.

[26] G. Suarez-Tangil, M. Conti, J. Tapiador, and P. Peris-Lopez, "Detecting targeted smartphone malware with behavior-triggering stochastic models," in *Proc. of 19th European Symposium on Research in Computer Security (ESORICS'14)*, 2014, vol. 8712, pp. 183–201.

[27] C. Shannon and W. Weaver, "The Mathematical Theory of Communication," 1963.

[28] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, 2008, pp. 51–62.

[29] G. Pék, B. Bencsáth, and L. Buttyán, "nEther: In-guest detection of out-of-the-guest malware analyzers," in *Proc. of the Fourth European Workshop on System Security (EUROSEC'11)*, 2011, pp. 3:1–3:6.

[30] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *J. Comput. Virol.*, vol. 7, no. 4, pp. 247–258, 2011.

[31] D. Babić, D. Reynaud, and D. Song, "Recognizing malicious software behaviors with tree automata inference," *Form. Methods Syst. Des.*, vol. 41, no. 1, pp. 107–128, Aug. 2012.

[32] M. J. Jurczyk. (2014, Nov.) Windows win32k.sys system call table. [Online]. Available: http://j00ru.vexillium.org/win32k_syscalls/

[33] J. R. Norris, *Markov Chains*. Cambridge University Press, 1998.

[34] M. J. Quinn and N. Deo, "Parallel graph algorithms," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 319–348, Sep. 1984.

[35] S. Edelkamp and R. E. Korf, "The branching factor of regular search spaces," in *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI'98 / IAAI'98)*, 1998, pp. 299–304.

[36] S. Maji, A. Berg, and J. Malik, "Classification using intersection kernel support vector machines is efficient," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'08)*, June 2008, pp. 1–8.

[37] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[38] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 8, pp. 832–844, Aug. 1998.

[39] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[40] A. Sami, H. Rahimi, and B. Yadegari, "Malware detection by behavioural sequential patterns," *Comput. Fraud and Secur.*, vol. 2013, no. 8, pp. 11 – 19, 2013.

[41] VirusTotal. (2015, Feb.) File types statistics. [Online]. Available: https://www.virustotal.com/en/statistics/

[42] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. of the 5th International Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'08)*, 2008, pp. 108–125.

[43] P. Vinod, V. Laxmi, M. Gaur, and G. Chauhan, "Momentum: Metamorphic malware exploration techniques using MSA signatures," in *Proc. of International Conference on Innovations in Information Technology (IIT'12)*, March 2012, pp. 232–237.

[44] D. Quist, L. Liebrock, and J. Neil, "Improving antivirus accuracy with hypervisor assisted analysis," *J. Comput. Virol.*, vol. 7, no. 2, pp. 121–131, May 2011.

[45] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod, "Behavioural detection with api call-grams to identify malicious PE files," in *Proc. of the First International Conference on Security of Internet of Things (SecurIT'12)*, 2012, pp. 85–91.

[46] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Lett.*, vol. 27, no. 8, pp. 861–874, 2006.

[47] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.

[48] H. Bunke, P. Foggia, C. Guidobaldi, and M. Vento, "Graph clustering using the weighted minimum common supergraph," in *Graph Based Representations in Pattern Recognition*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2726, pp. 235–246.

[49] Microsoft. (2015, Feb.) Kernel object. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms724485%28VS.85%29.aspx

[50] Boost-Software. (2015, Feb.) Graph library. [Online]. Available: http://sourceforge.net/projects/boost/files/boost/1.57.0/boost_1_57_0.tar.gz/download

[51] K. Kaczmarski, P. Przymus, and P. Rzazewski, "Improving high-performance GPU graph traversal with compression," in *New Trends in Database and Information Systems II*, ser. Advances in Intelligent Systems and Computing. Springer International Publishing, 2015, vol. 312, pp. 201–214.

[52] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010, pp. 399–412.

[53] Y. Wu and R. Yap, "Experiments with malware visualization," in *Proc. of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'13)*, 2013, vol. 7591, pp. 123–133.

[54] P. Trinius, T. Holz, J. Gobel, and F. Freiling, "Visual analysis of malware behavior using treemaps and thread graphs," in *Proc. of 6th International Workshop on Visualization for Cyber Security (VizSec'09)*, 2009, pp. 33–38.

[55] D. Quist and L. Liebrock, "Visualizing compiled executables for malware analysis," in *Proc. of 6th International Workshop on Visualization for Cyber Security (VizSec'09)*, Oct 2009, pp. 27–32.

[56] J. Saxe, D. Mentis, and C. Greamo, "Visualization of shared system call sequence relationships in large malware corpora," in *Proc. of the Ninth International Symposium on Visualization for Cyber Security (VizSec'12)*, 2012, pp. 33–40.

[57] S.-T. Liu, H. ching Huang, and Y.-M. Chen, "A system call analysis method with mapreduce for malware detection," in *Proc. of IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS'11)*, Dec 2011, pp. 631–637.