11-2013

A scalable approach for malware detection through bounded feature space behavior modeling

Mahinthan CHANDRAMOHAN

Hee Beng Kuan TAN

Lionel C BRIAND

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Bindu Madhavi PADMANABHUNI

A Scalable Approach for Malware Detection through Bounded Feature Space Behavior Modeling

Mahinthan Chandramohan[†], Hee Beng Kuan Tan[†], Lionel C. Briand[‡], Lwin Khin Shar[†] and Bindu Madhavi Padmanabhuni[†]

[†]School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore.
{mahintha001, padm0010}@e.ntu.edu.sg, {ibktan, lkshar}@ntu.edu.sg

[‡]SnT Centre, University of Luxembourg, Luxembourg.
lionel.briand@uni.lu

***Abstract*—In recent years, malware (malicious software) has greatly evolved and has become very sophisticated. The evolution of malware makes it difficult to detect using traditional signature-based malware detectors. Thus, researchers have proposed various behavior-based malware detection techniques to mitigate this problem. However, there are still serious shortcomings, related to scalability and computational complexity, in existing malware behavior modeling techniques. This raises questions about the practical applicability of these techniques.**

This paper proposes and evaluates a bounded feature space behavior modeling (BOFM) framework for scalable malware detection. BOFM models the interactions between software (which can be malware or benign) and security-critical OS resources in a scalable manner. Information collected at run-time according to this model is then used by machine learning algorithms to learn how to accurately classify software as malware or benign. One of the key problems with simple malware behavior modeling (e.g., *n*-gram model) is that the number of malware features (i.e., signatures) grows proportional to the size of execution traces, with a resulting malware feature space that is so large that it makes the detection process very challenging. On the other hand, in BOFM, the malware feature space is bounded by an upper limit *N*, a constant, and the results of our experiments show that its computation time and memory usage are vastly lower than in currently reported, malware detection techniques, while preserving or even improving their high detection accuracy.

***Index Terms*—Malware detection, Malware behavior modeling**

I. Introduction

Exponential growth of malware (malicious software) is a major threat in the software industry. Symantec, an anti-malware vendor, reported that more than 403 million new malware variants were created in 2011, a 41% increase over 2010 [25]. At the same time, targeted attacks, such as Stuxnet and Duqu, and Advanced Persistent Threats (APT) also showed a steady increase in recent years [25]. Despite the widespread use and availability of various anti-malware (commonly known as anti-virus) tools, the growth of malware is phenomenal. It is observed that malware has greatly evolved, where new malware has become very sophisticated and is designed to avoid traditional anti-virus signatures, using various obfuscation techniques [6].

Given the alarming growth of malware, a significant amount of research has focused on proposing various malware detection techniques to mitigate this problem. We can divide these techniques into two broad categories: static (or signature-based) and dynamic (or behavior-based) malware detection. Signature-based malware detection has an advantage over behavior-based analysis since it examines the static content of the malicious binary and thus, is able to achieve full code coverage. In addition, using signatures, it is even possible to detect malicious applications before they are executed. However, the major limitation of signature-based malware detection is that it can be easily evaded by basic obfuscation techniques. Further, malware authors can change the syntactic characteristics (i.e., structure) of a malicious program without changing its semantics (i.e., behavior) [4, 22]. Unfortunately, signature-based malware detection is still the predominant detection method today [20].

To overcome the limitations of signature-based malware detection, security researchers have proposed various behavior-based malware detection techniques [5, 8, 14, 19, 20] that focus on the semantics of the malicious application. In particular, these techniques examine the run-time behavior of the malicious binary and analyze the system calls (e.g., Win32 and native API functions) invoked during execution, in order to model its malicious behavior. The key behavior-based malware modeling techniques include: bags of system calls [5], sequence of system calls such as *n*-gram model [19, 5], individual system call analysis [14], behavioral graphs [8] and system call dependency graphs [20].

However, one of the key issues in existing behavior-based malware modeling techniques is that the scalability of these approaches is highly problematic. Here, scalability refers to the size of malware features (or signatures) extracted from an execution trace. For example, even a simple model such as bags of system calls, generates a number of malware features that grows proportionally to the size of execution traces [5]. This makes the detection process impractical as huge feature spaces make the learning process computationally intensive and detection might be negatively affected if most of these features are irrelevant. In addition to the scalability of feature spaces, unacceptably high computational complexity and memory consumption are also expected to impact the

practicality and efficiency of a malware detector. As a result, practical applications of complex malware behavior modeling techniques, such as system call dependency graph and behavioral graph, are very limited in practice. For example, in [20], it is reported that it took 12-48 hours to extract malware specifications from a network worm using a graph mining algorithm.

In this paper, we propose and evaluate a simple yet efficient malware behavior modeling technique—called *BOFM*—that systematically captures the interactions between malware and security-critical system resources in a scalable manner, at an adequate level of abstraction. In BOFM, scalability is achieved through constructing an upper-bounded malware feature space with a predetermined value *N* (discussed in Section IV). In other words, BOFM can extract malware features that do not grow in proportion with the number of program samples under examination. In addition, BOFM is resilient against basic obfuscation techniques [6].

As BOFM is both accurate and efficient, it can be used to complement traditional anti-virus tools by leveraging on BOFM features to accurately detect malware at the end host. Indeed, our results show that, when combined with machine learning techniques, BOFM is not only at least as accurate as existing malware detection techniques, but its computation times (e.g., for matching signatures) and memory consumption are vastly lower as well, thus making it a much more practical and scalable approach.

In summary, this paper makes the following contributions:

- We propose a malware behaviour modelling technique (called BOFM) that captures malicious interactions between malware and security-critical OS resources in a scalable manner. This is in turn combined with machine learning techniques for automated malware detection.
- We conducted and report an experiment involving 5300 malware and 100 benign samples collected from various sources. Our experimental results show that BOFM, when combined with appropriate machine learning classifiers, can achieve 99.4% malware detection rate with no false positives, the latter being important in our context.
- We show that the feature space generated by BOFM is of fixed dimension, does not grow with the number of malware samples under examination, and is three orders of magnitude smaller than with the best reported techniques for malware behaviour modelling. As a result, computation times and memory usage for extracting program features and malware detection are vastly decreased.

The paper is organized as follows. Section II summarizes the related work and our motivation. Section III gives an overview of our behavior modeling technique. Section IV defines our bounded feature space behavior modeling technique. Section V describes the feature vector construction. Section VI presents the detection method. Section VII describes the experimental design. Section VIII summarizes and discusses the experimental results. Finally, Section IX concludes the paper.

II. Related Work and Motivations

Past research includes several useful techniques for malware detection. In this section, we shall present related research works and discuss the pros and cons of these approaches.

The recent study on behavior-based malware detection reported by Canali et al. [5] is most relevant to our current work. In [5], the authors have done a comprehensive study on analyzing the efficiency of various malware behavior modeling techniques. They organized the behavior models based on three dimensions: (1) the granularity of basic model elements (i.e., system calls at various levels of abstraction, such as with and without parameters), (2) the relationship among basic model elements, such as *n*-gram, *m*-bag and *k*-tuple[1] and, (3) cardinality of each specification (i.e., values of *n*, *m* and *k*). A series of more than 200 experiments were conducted on a large set of malware and benign samples to systematically identify the optimal behavior model for malware detection. Their experiment results revealed that the optimal behavior model, *2-bags of 2-tuples of action with arguments*, achieved a 99% detection rate with only a 0.4% false positive rate.

Though the above results are very encouraging, there are several shortcomings associated with this approach. The most important issue is that the scalability of this approach is highly problematic. The behavior modeling techniques such as *n*-gram, *m*-bag and *k*-tuple can easily generate huge feature spaces. For example, from an execution trace with 400 unique system calls, more than 10 million features are constructed using *m*-bag model with a bag cardinality of 3 (*m*=3) . This problem is even worse in tuple-based behavior models. In [5], it is also reported that the memory consumption is one of the major threat to practical implementation of this approach, where on a standard machine (Intel Dual Core 2.66 GHz with 4GB of RAM) the prototype malware detector consumed 1GB of RAM for 5 million malware signatures. Further, it is also reported that feature extraction from malware samples is itself a computationally intensive task, where almost 2 days of computation are required for *tuples of system calls with arguments* model.

In addition, the approach in [5] requires several parameter tunings for optimal performance that makes it less practical. Moreover, it is observed that intensive parameter tuning is often associated with overfitting problems. For example, to manage the huge feature space, authors have proposed a feature pruning mechanism. A malware feature is discarded if it is not general enough (i.e., doesn't detect at least five malwares) or too redundant (i.e., not represented by 20,000 other signatures), where the values 5 and 20,000 are arbitrarily selected. Further, there is a trade-off in selecting the cardinality of a specification (e.g., number of basic model elements in a bag), where increasing the cardinality may result in a high detection rate but also lead to overfitting. For example, in [5], it is reported that detection rates achieved by tuple-based behavior models are highly sensitive to

[1] Please refer to [5] for definitions of *n*-gram, *m*-bag and *k*-tuple models

cardinality. Finally, the alert threshold, the number of malware signatures that need to be matched to flag an unknown program as malware, significantly influences the detection rate and false positive rate. In [5], it is shown that small alert threshold results in high detection and false positive rate and vice versa. Thus, choosing the appropriate values for these parameters is crucial but complex in practice.

Apart from [5], there are several other behavior-based malware modeling techniques proposed in the literature. Lanzi et al. [19] proposed AccessMiner, a malware detection technique based on system-centric malware models, where the interaction between benign sample and the OS resources is modeled in a system-centric manner. This addresses the limitation of program-centric approaches. Further, authors empirically proved the inefficiencies of *n*-gram based malware behavior modeling. In addition to generic malware detection [15, 13], system call and library call based behavior modeling is also proposed to detect more specific class of malware, such as spyware (Kirda et al. [21]) and botnets (Stinson et al. [16]) detection.

Kolbitsch et al. [8] proposed an efficient and effective malware detection approach at the end host, where it models the malware behavior as a graph and detection is done at the end host using graph matching. Similarly, Fredrikson et al. [20] proposed malware specification mining using dependency graphs. They managed to achieve a higher detection rate than two commercial behavior-based malware detectors. However, graph mining still remains very computationally intensive, where it is reported that it took around 12-48 hours to extract malware specification from certain network worms. In addition, the dataset used in most of these approaches are very limited. For example, Kolbitsch et al. [8] used only 563 malware samples and 10 benign samples, similarly, Christodorescu et al. [13] used 16 malware and 6 benign samples, Stinson et al. [16] used 6 malware and 9 benign samples, and Martignoni et al. [15] used only 7 malware and 6 benign samples for evaluation.

We derive the following key observations based on the abovementioned, behavior-based malware detection approaches [5, 19, 8, 13, 16, 15]:

- The practicality and efficiency of malware detection techniques are characterized on four dimensions: the size of feature space, computational complexity, overhead in terms of additional pre-processing activities, and detection accuracy.
- Simple malware behaviour models such as *n*-gram, *m*-bag and *k*-tuple, generate huge feature spaces and require various pruning and parameter tuning mechanisms to alleviate the problem.
- More complex malware behaviour models, such as dependency and behavioural graphs are, generally, highly computationally and data intensive.

In the next two sections, we explain our proposed approach, **Bo**unded **F**eature space behavior **M**odeling (BOFM), which aims at making malware detection scalable, efficient, and practical, while retaining or improving the high accuracy reported thus far in the literature.

III. Overview

Malwares usually achieve their objectives through performing malicious actions on security-critical, operating system resources. An ***action*** corresponds to a high-level operation (e.g., reading a file) that is composed of a set of related system calls to achieve an externally meaningful objective [3, 5]. For example, reading a file may require two system calls: (1) NtOpenFile to open the file, and (2) NtReadFile to read the file content. The main advantage of using actions over system calls is that different versions of the same operating system (e.g., Windows 2000 and Windows XP) may use different names for system calls that are in fact serving the same purpose [3] and, as a result, analyzing system calls directly may result in dealing with unnecessarily large amounts of data. For example, in Windows OS, the system calls NtCreateProcess and NtCreateProcessEx are both used to create a new process. Thus, these two system calls can be mapped to a single action called CreateProcess. System call sequences can be mapped to actions using a mapping algorithm [7]. There are a number of different mapping algorithms used in the malware research community. Depending on the algorithm used, the system call sequence ⟨ NtOpenFile, NtReadFile ⟩ can be mapped to two distinct actions OpenFile and ReadFile, respectively, or both system calls can be combined to represent a single action ReadFile.

In the remainder of this section, we first discuss the various types of OS resources and then, we introduce and illustrate our behavior modeling technique by using an example.

A. Operating System Resource Types

The analysis of malicious behavior is usually carried out through examining the actions it performs on security-critical system resources. In the literature, researchers usually model malware behavior based on its interaction with certain types of OS resources, such as file system, registry, process and network. For example, Kolbitsch et al. [8] considered security-relevant system calls associated with file system, registry, network, process and system services for malware detection, whereas in our previous work [9], we considered the actions performed on four security-critical OS resource types such as file system, registry, process and network, for malware clustering. Based on the broad classification of system calls reported in [28] and Windows OS internals [10], we have considered the following OS resource types in our study: file system, registry, process, thread, section, network and synchronization. Next, we shall briefly describe each of these OS resource types.

- **File System.** Operating system and the programs that run on it are made up of individual files. A file is an instance of any opened file or I/O device.
- **Registry.** Registry is a system-defined database in which applications and system components store and retrieve configuration data [29].
- **Process and Thread**. Process is the virtual address space and control information necessary for the exe-

cution of a set of threads One or more threads run in the context of such process [27].

- **Network.** This corresponds to the network related activities of the program being executed.
- **Synchronization**. This aims to protect shared resources from simultaneous access by multiple threads or processes [31].
- **Section.** Section represents a portion of memory that can be shared, where a process can use section to share parts of its memory address space with other processes [32].

We model malicious behavior based on the sets of actions that malware performs on individual OS resource instances. An ***OS resource instance*** corresponds to an identifier (or instance) of an OS resource type. For example, for File System, file names (e.g., C:\foo.exe and C:\Windows\abc .dll) are identifiers and the actions performed on each of these file instances include OpenFile, ReadFile, and DeleteFile. A comprehensive list of actions that a malware can perform on each OS resource type is given in Table 1.

TABLE 1: OS RESOURCES AND CORRESPONDING ACTIONS

OS Resource Types	# of Actions	List of Actions
File system	14	CreateDirectory, QueryDirectory, CreateFile, SetFileInformation, UnLockFile, LockFile, OpenFile, WriteFile, QueryFileAttributes, QueryFileVolume, DeleteFile, ReadFile DeviceControl, QueryFileInformation
Registry	7	CreateKey, DeleteKey, DeleteValue, SetValue, OpenKey, NotifyChangeKey QueryValue
Process/Thread	6	SetInformationProcesses, Create-Process, CreateThread, OpenProcess, KillProcess, QueryInformationProcess
Synchronization	6	CreateMutex, OpenSemaphore, CreateSemaphore, OpenMutex, ReleaseMutex, ReleaseSemaphore
Network	1	NetworkConnection
Section	4	OpenSection, CreateSection, QuerySection, MapViewOfSectoin

Next, we shall present an example to illustrate our behavior modeling technique. This example is used as a running example in this paper.

B. Example

A sample malware execution trace is given in Figure 1, where system calls are already mapped to high-level actions. In a real world scenario, a single malware execution trace can contain several thousands of actions. However, to keep it simple, we have only considered few file and registry related actions in this example. The behavior of our pseudo malware is given below:

- Creates a malicious executable along with three other dummy files.
- Reads two system files and a dummy file several times.
- Creates a registry key and sets its value.
- Deletes all the dummy files.

```
 1: CreateFile("C:\Windows\malicious.exe")
 2: CreateFile("C:\Windows\…\dummy1.txt")
 3: ReadFile("C:\Windows\…\dummy1.txt")
 4: CreateFile("C:\Windows\dummy2.dll")
 5: ReadFile("C:\Windows\…\sysfile1.ini")
 6: ReadFile("C:\Windows\…\sysfile2.dll")
 7: CreateKey("HKLM\Software\…\…\key")
 8: SetValue("HKLM\Software\…\…\key", value)
 9: ReadFile("C:\Windows\…\dummy1.txt")
10: ReadFile("C:\Windows\…\dummy1.txt")
11: DeleteFile("C:\Windows\…\dummy1.txt")
12: CreateFile("D:\Personel\dummy3.exe")
13: DeleteFile("D:\Personel\dummy3.exe")
14: DeleteFile("C:\Windows\dummy2.dll")
15: ReadFile("C:\Windows\…\sysfile2.dll")
```

Fig.1. Sample malware execution trace

TABLE 2: EXTRACTED MALWARE FEATURES

Id	Features (Action set)	OS Resource Instances
1	{CreateKey, SetValue}	HKLM\Software\…\…\key
2	{CreateFile}	C:\Windows\malicious.exe
3	{CreateFile, ReadFile, DeleteFile}	C:\Windows\…\dummy1.txt
4	{ReadFile}	C:\Windows\…\sysfile1.ini, C:\Windows\…\sysfile2.dll
5	{CreateFile, DeleteFile}	C:\Windows\dummy2.dll, D:\Personel\dummy3.exe

Table 2 shows in column 2 the five extracted features from the sample malware execution trace shown in Figure 1. As we will see in the next section, features are action sets, that is, features are constructed by grouping related actions performed by malware on individual OS resource instances, where related actions refers to actions belonging to the same OS resource type. Column 3, in Table 2, shows the OS resource instances on which features are performed. For example, the action set {CreateFile, ReadFile, DeleteFile} is performed on a file resource C:\Windows\…\dummy1.txt (*Id-3*) and action set {ReadFile} is performed on two different OS resource instances C:\Windows\…\sysfile1.ini and C:\Windows\…\sysfile2. dll (*Id-4*). In Table 2, *Id 2-5* represent malware features corresponding to File System and *Id-1* represents a feature corresponding to Registry. It is important to note that OS resource instances (column 3 in Table 2) are only used to identify related actions and are not included in the feature vectors used to support malware detection (Section V). This is due to the fact that malware tend to use random file names, mutex values and registry key values each time they execute and, therefore, there is no agreed-upon mechanism to generalize these highly volatile artifacts [19].

Next, we shall precisely define our malware behavior modeling technique, *BOFM*, and explain its properties.

IV. Bounded Feature Space Behaviour Modelling (BOFM)

A malware perform various actions on one or more OS resource instances. In the proposed BOFM, for each type of OS resources, the set of related actions performed by malware on an individual resource instance constitutes a ***feature*** of the malware. That is, in our example (see Section III.B), the set of related actions: {CreateKey, SetValue}, performed by malware on a registry instance: HKLM\Software\…\…\key, constitutes a feature (*Id-1*). In total, five features are extracted (see column 2 in Table 2) from the malware execution trace shown in Figure 1.

Due to our modeling preference, BOFM features hold the following three key properties;

Property 1: Regardless of the number of times an action is performed, if the same set of actions is performed on OS resource instances of the same type, this leads to identical malware features. For instance, in our example, ReadFile action is performed only once on file instance C:\Windows\…\sysfile1.ini and twice on file instance C:\Windows\…\sysfile2.dll; however, these two behaviors are considered to be identical and are represented by a single malware feature {ReadFile} (*Id-4*).

Property 2: The sequence, in which the actions are performed, by malware, is ignored in feature construction. That is, in our example, malware read the contents of system file: C:\Windows\…\sysfile1.ini way ahead of creating file: D:\Personel\dummy3.exe, however, this information (i.e., sequence) is not captured by BOFM features. In addition, ordering of actions in an action set is also ignored. That is, features {ReadFile, QueryFileInformation} and {QueryFileInformation, ReadFile} are considered identical.

Property 3: Identical action sets which are performed on two different OS resource instances of same type are modeled as a single feature. In our example, action set {CreateFile, DeleteFile} is modeled as a single malware feature even though it is performed on two different file resource instances; C:\Windows\dummy2.dll and D:\Personel \dummy3.exe (*Id-5*).

Next, we shall formally derive the upper-bound for malware features constructed using BOFM.

Upper-bound. Let us assume there are n types of OS resources and for each OS resource type j ($1 \leq j \leq n$), the possible actions that a program can perform are always predefined and fixed. The list actions considered for each OS resource type in Table 1 is a representative example. Thus, the total number k_j of possible actions that a malware may perform on a resource instance of type j is a constant. Consequently, the maximum number m_j of possible features (or action sets) with regard to OS resource type j is also a constant and can be computed as follows:

$$m_j = C_1^{k_j} + C_2^{k_j} + C_3^{k_j} + \ldots + C_{k_j}^{k_j} \qquad (1)$$

Where, $C_h^{k_j} = \frac{k_j!}{h!(k_j-h)!}, (1 \leq h \leq k_j)$. Therefore, as a result of applying BOFM, the total number of possible features N, extracted from all resource types, is always the following constant:

$$N = \sum_{j=1}^{n} m_j \qquad (2)$$

From equation (2), it can be seen that the total number of features of a malware depends only on the total number of OS resource types and the number of possible actions performed on each OS resource type.

Based on the number of actions considered for File System (Table 1), using equation (1), the maximum possible number of malware features that can be extracted is $C_1^{14} + C_2^{14} + \cdots + C_{14}^{14} = 16{,}383$. Similarly, the maximum possible number of malware features extracted for various resources are: registry: 127, network: 1, process/thread: 63, synchronization: 63 and section: 15. Finally, using equation (2), the total number of malware features (*N*) that can be extracted from these six OS resource types sums up to 16,652. It's worth noting that the value of *N*, calculated above, is specific to this study. In practice, it will vary based on the number of OS resource types and list of actions considered in a given context.

Hence, in contrast to existing approaches in which the feature space grows in direct proportion to number of malware instances under examination, the total number of possible features for malware detection under our approach has an upper bound N (*N*=16,652). That is, our approach has a bounded feature space and this is expected to improve the scalability of malware behavior modeling. Further, we observed that malwares often perform a combination of actions that are not normally performed by benign applications. This behavior is captured by the hypothesis below.

Hypothesis (Action set characterization). To achieve malicious objectives, malware tend to perform sets of actions, on a number of OS resource types that are significantly different from benign applications.

Rationale. Obfuscation techniques, such as dead-code injection, subroutine reordering, instruction substitution and code transposition, are widely used to evade traditional, signature-based malware detection [6]. It is observed that malware authors often: (1) reorder independent actions[2] [13], and (2) repeat certain actions many times [30], e.g., perform ReadFile action in a loop, to break the byte sequence (or code pattern) without affecting the semantics of the program to fool byte (or action) sequence-based malware detection techniques. Therefore, accounting for the sequence of actions in behavior modeling may have the adverse effect of failing to capture identical, high-level behavior with different action sequence. In addition, as opposed to sets, sequences of actions, similar to

[2] These actions are independent from others and any permutation of these actions will lead to the same end behavior [13]

n-gram, would result in a huge feature space and thus, would be much less scalable. Thus, we modeled the malware behavior as a *set* of actions, in contrast to sequences, to overcome the above mentioned obfuscation techniques. The advantage of using set in malware behavior modeling is twofold: (1) repeated actions are ignored (*property 1*), and (2) agnostic to reordering of independent actions (*property 2*). Though it is noted that there is a trade-off in using sets as the ordering of actions may constitute valuable information, in practice, malware authors often use obfuscation techniques that render this information useless. Further, OS resource instances (column 3 in Table 2) are not considered for malware detection as they are highly volatile (i.e., involve randomness) and there is no agreed-upon mechanism to generalize these highly volatile artifacts such as file names, mutex values and IP addresses [19] (*property 3*).

V. Construction of Feature Vectors

In this section, we explain how we extract malware features from execution traces and embed them in feature vectors.

A. Collecting Execution Traces

The run-time behavior of malware instances are monitored using a Sandbox, which is a dynamic malware analysis tool such as CWSandbox [1], Anubis [2], and Cuckoo Sandbox [12]. These systems execute programs in a controlled environment, monitor their behavior, and generate behavior reports. These reports generally contain high-level, action based malware behavioral characteristics [1] such as newly created/modified/deleted file details, registry keys, and network traffic details. Few sandboxes, such as Cuckoo sandbox [12], also provide low-level behavioral characteristics (i.e., Win32 and native API functions based). In such cases, system calls can be mapped to relevant high-level actions using an appropriate mapping algorithm [7]. It is also noted that system calls can be used to model malware behavior but, as mentioned earlier, one must be aware of different system calls serving the same purpose (e.g., NtCreateProcess and NtCreateProcess-Ex).

B. Extraction of Features

Feature extraction from an execution trace involves three steps. They are as follows:

Step 1: OS resource instances present in the execution trace are identified.

Step 2: Related actions[3] corresponding to an OS resource instance are grouped, forming action sets.

Step 3: Repeat Step 2 until all the OS resource instances identified in Step 1 are covered.

Each unique action set, constructed in Step 2, constitute a feature (see Section III.B). Similar to other approaches [9, 11], we use feature vector to embed the extracted malware features. Next, we shall explain feature vector construction.

[3] Related actions refer to actions belonging to the same OS resource type.

Features / Samples	{CreateKey, SetValue}	{CreateFile}	{CreateFile, ReadFile, DeleteFile}	{ReadFile}	{CreateFile, DeleteFile}	…	$F_{16,652}$
Malware 1	1	1	1	1	1	0	0

Fig. 2. Feature vector

C. N-dimensional Feature Vector

For each malware, once the malware features are extracted (column 2 in Table 2) from the execution traces, we embed them in an *N*-component (or *N*-dimensional) feature vector, where *N* is the total number of possible features (equation 2). Each component in a feature vector is designated to represent a possible malware feature. We re-emphasize that OS resource identifiers are only used to identify related actions and are not part of feature vectors. Figure 2 depicts a feature vector constructed using malware features (column 2 in Table 2) extracted from execution traces shown in Figure 1. In a feature vector, the n^{th} component represents feature *n* (denoted by F_n) and each feature is assigned a value '1' or '0' to denote the presence (i.e., *active*) or absence of that feature, respectively. 'Active' features[4] refer to those features present in a malware or benign program.

It is also noted that extracting features from benign programs exactly follows the same process as explained above, except that it is executed on a real-world machine instead of a sandbox to collect representative execution traces. In addition, the length of benign feature vectors is also upper-bounded by a constant *N* (equation (2)). To summarize, each malware and benign application is converted into an *N*-dimensional binary feature vector, in our case *N*=16,652.

VI. Detection Method

Based on our analysis of related work (Section II), we find that not many behavior-based malware detection frameworks adopt Machine Learning classification techniques to build detection engines. This is may be due to the fact that when the feature space is extremely large, the learning process will be computationally intensive and negatively affected if most of the features are irrelevant. In contrast, due to its limited feature space, BOFM is amenable to the use of Machine Learning (ML) classification techniques for building Malware Detection models. In our approach, Machine Learning classifiers are used to classify unknown software as either malware or benign. We tried and compared a number of ML classification techniques and report here on the results with Logistic Regression (LR) [18] and Support Vector Machine (SVM) [23]. The latter is reported as it is a recent technique, that has shown to work well in a number of applications, is based on a non-probabilistic theory about learning structures in data, and yields the best results in our particular case. The former is a standard probabilistic technique that yields regression models

[4] In the following sections of the paper, when we refer to 'features' extracted by BOFM, we implicitly refer to 'active' features, unless otherwise stated. The number of active features is always less than or equal to *N*.

which are easy to interpret; in particular to assess what features are statistically significant predictors of malware.

The malware detection process includes two phases: model building and model evaluation. During the model building phase, a training-set of benign and malware feature vectors is used by the ML algorithm to build a classifier. The model evaluation phase aims at assessing the classifier accuracy in a realistic fashion. By analyzing the actual classification (benign or malware) of feature vectors present in the training-set, an ML algorithm generates a trained classifier, e.g., a logistic regression equation with estimated parameter values. Next, during the model evaluation phase, a test-set, of benign and malware feature vectors, is classified by the trained classifier. Based on classification results, the performance of the classifier is evaluated by computing standard accuracy evaluation criteria. Note that computing such criteria requires the actual class labels of feature vectors in the test-set in order to compare the actual class with the class predicted by the trained classifier [24].

VII. Experimental Design

To evaluate the effectiveness of BOFM in distinguishing between malicious and benign behaviors, we performed a large set of experiments. In the following sub-sections, we describe the datasets used in our experiments and describe the evaluation criteria used to evaluate our malware predictions.

A. Experimental dataset

We used three different datasets in our experiments. The first dataset (called malware-train) is a collection of execution traces of 5,000 malware samples obtained from the Anubis [2] database. Similarly, the second dataset (called malware-test) contains execution traces of 300 malware samples that are collected on a machine that is not used to run Anubis [2]. It is noted that these two datasets, malware-test and malware-train, are obtained from Canali et al. [5]. The final dataset (called benign) contains the execution traces of 100 benign samples collected from five different real-world machines (PCs), where each machine ran 20 benign programs. Obtaining execution traces from various machines, for both benign and malware programs, helps lower the chances that machine specific artifacts influence the final outcome [5].

The benign dataset consists of applications that are commonly used by a standard user (e.g., MS Word, Firefox) and that are mostly interactive applications. Thus, following the approach in [20], we obtained representative execution traces by simulating the user interaction with the application for around 15-20 minutes. For example, for a word processor application (i.e., MS Word), we created a new document including text, images and diagrams and saved it to disk, used several MS word plug-ins, such as EndNote, and opened few already existing documents. Similarly, for a web browser (e.g., Firefox,), we connected to our university webpage (www.ntu.edu.sg) and downloaded few documents from the site, uploaded few files the server, sent an e-mail using Gmail, and used several browser plug-ins such as adBlock and PDF Viewer. Likewise, for all interactive benign applications, we performed a set of required operations to get representative execution traces. It is important to note that to be consistent with malware execution traces obtained using Anubis [2], we selected the same subset of system calls, as used in Anubis [2], from the benign execution traces.

In addition, to ascertain whether and explain why a 15-minute simulated user interaction (with a benign application) is good enough to represent real-world user behavior in terms of BOFM features, we conducted a simple experiment. We compared a representative execution trace of MS Word application (monitored for eight hours) obtained from a real-world machine with an execution trace of same application but with simulated user interaction. The findings are summarized in Table 3.

From Table 3, it can be seen that the execution trace obtained using real-world user interactions (trace *A*) is larger in size and has a larger number of system calls than the execution trace with simulated user interaction (trace *B*). However, the number of features extracted using BOFM is larger for trace *B* than for trace *A*. This shows that in real-world, people often tend to use a small fraction of operations (e.g., intuitive operations) and thus generate large execution traces with limited number of useful features. Hence, in order to get real and representative execution traces for an application, and thus build accurate malware predictors, one should simulate the user interaction to explore as many features as possible.

TABLE 3: COMPARISON OF EXECUTION TRACE FOR SIMULATED USER AND REAL-WORLD USER INTERACTIONS

	Real-world User Interaction (*A*)	Simulated User Interaction (*B*)
Application	MS Word	MS Word
Execution environment	Real-world PC	Real-world PC
Execution time	8 hours	15 minutes
Size of execution trace	129,108 KB	27,313 KB
# of system calls obtained	955,463	180,614
Extracted BOFM features	77	108
# of common features	57 (74%)	57 (52.78%)

Further, when selecting the benign applications for evaluation, we made sure that they were from one of the following functionally diverse categories: word processors (e.g., MS Word), text editors (e.g., MS Wordpad), command-line shell (e.g., cmd.exe), web browser (e.g., Firefox), file transfer (e.g., Filezilla FTP server), remote access (e.g., Putty ssh client), e-mail (e.g., MS Outlook), IDE (e.g., MS Visual Studio), media (e.g., VLC player), game (e.g., Chess Titans), anti-visus tool (e.g., AVG antivirus), VOIP (e.g., Skype), cloud storage (e.g., Dropbox), reader (e.g., Adobe PDF reader) and other utility tools (e.g., Google Desktop). We observed that execution traces of benign applications in the same category looked similar. For example, using BOFM, we managed to extract 66 features from 'Notepad' and 73 features from 'WordPad' execution traces, out of which 63 features were common to both applications. That is, 95.5% of Notepad features and 86.3% of WordPad features are identical. This indicates that, as long as we include a few benign applications

from each category, it is sufficient to cover a wide range of benign applications in terms of functionality. Thus, we can confidently say that our benign dataset of 100 applications (at least three from each category) is sufficiently representative to build classifiers to accurately predict malware, as it will be confirmed by our experimental results. It is also worth noting that in the literature, researchers have used very small benign dataset in the range of 5-18 applications [5] and that our experiment is much more extensive with that respect.

Finally, as mentioned in Section VI, malware detection consists of two phases: model building and evaluation. To build the models, we used a training-set consisting of malware-train and execution traces of benign samples obtained from 4 (out of 5) machines. To evaluate the classifier models, we used a test-set consisting of malware-test and benign execution traces from the machine that was not used for training. This process is repeated five times where, on each iteration, the test-set consists of benign execution traces selected from a different machine. Due to space constraints, only averages, across the five experiments, are presented in the paper. Table 4 presents an overview of the benign and malware datasets used in our experiments.

TABLE 4: OVERVIEW OF MALWARE AND BENIGN DATASETS

	Malware Dataset	Benign Dataset
Total number of samples	5,300	100
Execution environment	Sandbox [11]	Real-world PCs
Max. size of an execution trace	37,416 KB	320,967 KB
Min. size of an execution trace	17 KB	446 KB
Avg. size of an execution trace	527 KB	56,544 KB
Total no. of system calls	31,506,686	52,447,089
Avg. # of system calls/sample	5,945	524,471

B. Evaluation Measures

We employ standard evaluation measures, including *True Positive Rate (TPR)*, *False Positive Rate* (*FPR*) and *Total Accuracy*, to evaluate the malware detection accuracy. We refer to definitions in [24] for further details but for the sake of completeness, we briefly explain them here. We can use the following contingency table to define the four possible outcomes (i.e., *TP*, *FP*, *FN* and *TN*) from a binary classifier.

		Actual Value: **Malware**	Actual Value: **Benign**
Predicted Outcome	**Malware**	True Positive (*TP*)	False Positive (*FP*)
	Benign	False Negative (*FN*)	True Negative (*TN*)

True Positive Rate (*TPR*) is in our context the proportion of malware samples correctly classified as malware. Similarly, False Positive Rate (*FPR*) is the proportion of benign samples misclassified as malware. Finally, *Total Accuracy* (or detection accuracy) measures the overall proportion of correctly classified instances, either malware or benign. These measures are formally defined as follows:

TABLE 5: SUMMARY OF MALWARE DETECTION ACCURACY ACHIEVED BY LR AND SVM

Classifier	True Positive Rate/Counts	False Positive Rate/Counts	Total Accuracy/Counts
LR	0.996/1494	0.01/1	0.996/1593
SVM	0.994/1491	0.00/0	0.994/1591

$$TPR = \frac{TP}{TP + FN} \quad (3)$$

$$FPR = \frac{FP}{FP + TN} \quad (4)$$

$$Total\ Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (5)$$

C. Experimental Setup

All the experiments were executed using a 4 core Xeon(R) with 4 GB RAM machine installed with Ubuntu 12.04. In addition, a well known machine learning tool, WEKA [17], was used to build the malware detection classifiers.

Next, we shall present and discuss the malware detection accuracy obtained by our framework.

VIII. EXPERIMENTAL RESULTS

To assess the accuracy of our classifiers in detecting malicious code, we measured the True positive Rate, False Positive Rate and Total Accuracy achieved by BOFM for both logistic regression (LR) and Support Vector Machine (SVM). As explained in Section VII.A, we used a training-set of 5000 malware and 80 benign samples and a test-set of 300 malware and 20 benign samples. The experiment is repeated five times with different sets of benign samples in the test-set (refer Section VII.A for explanation), following a standard 5-fold cross validation process.

The average results for LR and SVM, across all five experiments, are presented in Table 5 in the form of both rates and counts. From Table 5, it can be seen that both LR and SVM achieved similar detection accuracy. For LR, the slightly better detection accuracy of 99.6%, compared to 99.4% for SVM, comes at the cost of a 1% false positive rate. In malware detection, a lower (or zero) false positive rate is desired since the consequences of flagging a benign application as malware can be disastrous. For example, if a benign system file is flagged as malicious and deleted from the system, in the worst case, it may prevent the system from booting. Thus, we can conclude that SVM is a preferable solution to LR in our context.

Since our original test-set contained a much higher proportion of malware, we needed to check this imbalance did not bias our results and ran the experiment again with a randomly selected, balanced subset (test-set 2). Test-set 2 consists of 20 randomly selected (from a pool of 300 samples) malware samples and the 20 benign samples used in the original test-set. Again, we repeated the experiment five times following the same procedure as on the original data set and the averages across the five experiments were analyzed. SVM yielded a perfect accuracy of 100%, thus not misclassifying a

single program. However, LR achieved 99.5% detection accuracy with 1% false positive rate. Thus, based on these two experimental results, we can confirm that SVM performs relatively better than LR for malware detection. Further, the results also suggest that the accuracy of our overall approach based on BOFM is very encouraging.

In addition, we also performed sensitivity analysis to investigate how individual OS resource types influence malware detection rates. That is, we repeated the above experiment six times, using test-set 2, where on each instance we considered features corresponding to an individual OS resource type. For sensitivity analysis, we used SVM as our classifier and Table 6 summarizes the outcomes. From Table 6, it can be seen that for each OS resource type the false positive rate is unacceptably high. However, two OS resource types: file system and process/thread, managed to achieve a detection accuracy of 85% or greater with a false positive rate of 30% and 5%, respectively. Thus, we can conclude that features corresponding to individual OS resource types alone not sufficient for malware detection as they generate unacceptably high false positive rates.

A. Feature Space Analysis

In this section, we compare the features extracted from malware samples using BOFM and *n*-gram (n=2 or bigram) model, a simple behavior modeling technique. For this analysis, we considered 5000 malware samples that is used to train our classifier. One of the key aspects of BOFM is the bounded feature space, where the number of active features doesn't grow in proportion with the number of samples (or size of execution trace) under evaluation. To visualize this characteristic of BOFM, in Figure 3(b), we have plotted the feature space or active features' growth against malware sample sizes. From Figure 3(b), it can be seen that, for BOFM, the curve flattens as the number of malware samples increases. However, for bigram model (Figure 3(a)), the number of features grows proportionally with the number of malware samples under examination. It is also noted that a similar feature space growth trend is observed in other behavior modeling techniques such as *n*-bag and *k*-tuple. This illustrates the scalability of BOFM, against simple malware behavior models, in terms of feature space.

In addition, in order to investigate whether our original hypothesis holds, we analyzed how benign and malware program behaviors differ in general. In Table 7, we summarized the number of common features as well as the unique features corresponding to malware and benign datasets. From Table 7, it can be seen that around 51% and 43% of the features appear to be common in benign and malware samples, respectively. Having 57% of unique malware features strongly supports our hypothesis and suggests that behavior-based malware analysis is a promising approach to detect malicious software. Next, we shall briefly analyze some of the interesting malware and benign features

B. A Brief Analysis of Interesting Features

Through our analysis, we find that certain actions drove malware predictions to a large extent. To be more specific,

TABLE 6: SUMMARY OF SENSITIVITY ANALYSIS

OS Resource Type	Total Accuracy	FP Rate
File system	0.850	0.30
Section	0.750	0.50
Network	0.500	1.00
Synchronization	0.750	0.50
Process/Thread	0.975	0.05
Registry	0.750	0.50

TABLE 7: SUMMARY OF FEATURE SPACES

	Malware Dataset	Benign Dataset
Common features	243 (42.71%)	243 (50.63%)
Unique features	326 (57.29%)	237 (49.37%)
Total features	569 (100%)	480 (100%)

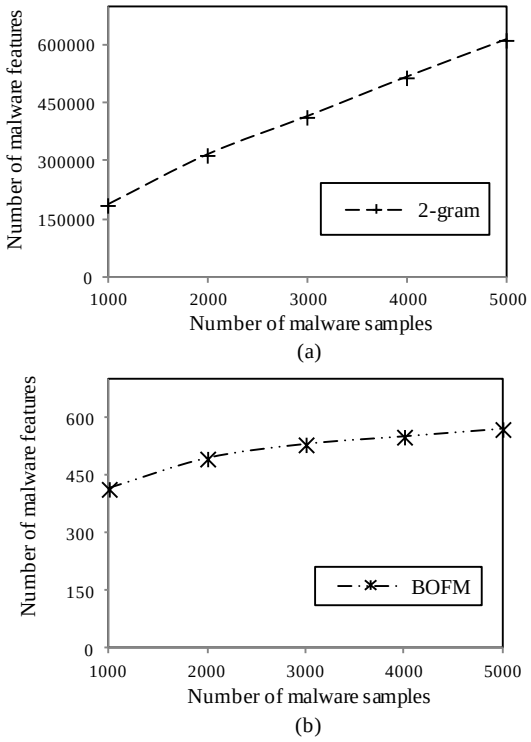

Fig. 3. Feature space growth of malware samples: (a) using bigram (n=2) model, and (b) using BOFM

NotifyChangeKey action (i.e., action that allows the running application to request notification for a registry key change [29]) is very widely used by malware samples when compared to benign samples. That is, around 86% (4,311/5,000) of malware applications performed NotifyChangeKey action on registry resource whereas only 15% (15/100) of benign applications performed it. Further, DeleteKey and DeleteValue actions (i.e., actions that delete keys and values from the registry, respectively) also played a significant role in distinguishing malware from benign applications. This behavior is expected as registry contains the system configuration settings and malware often create (or modify) registry keys and values to maintain persistence on the

infected system, allowing the malware to survive reboots. For example, modification to the registry key: HKLM\SOFTWARE \Microsoft\Windows\CurrentVersion\Run, allows the malware to automatically run every time when the Windows is started.

In addition, with regards to file resource, we find that there are several features such as {CreateFile, SetFileInformation, DeviceControl, ReadFile, WriteFile} that widely appeared in malware samples. Similarly, several features corresponding to process/thread resource predominantly appeared in malware samples. For example, {CreateThread, QueryInformation-Process, SetInformationProcess} feature appeared in 67.2% (3361/5000) of malware samples and only 4% (4/100) of benign applications. However, several other features such as {OpenFile, QueryAttributesFile} and {CreateMutex, ReleaseMutex} seemed to be too common, where they appeared in almost all the malware and benign applications.

C. Comparison with Canali et al.

As discussed in Section II, Canali et al. [5], in their recent study, proposed several malware behavior modeling techniques for malware detection. Their experiment results revealed that the optimal behavior model, *2-bags of 2-tuples of action with arguments*, achieved a 99% detection rate with a false positive rate of 0.4%. Using SVM to build a classifier, BOFM achieved 99.4% detection accuracy with no false positives and therefore improves the already accurate results of Canali et al. [5]. Since our benign sample is different from theirs (due to privacy reasons), this result should be interpreted with care but is nevertheless encouraging. More importantly, such improvement is obtained despite a dramatic size reduction in the malware feature space, as discussed next.

Canali et al. [5], on average, generated more than one million malware features (i.e., signatures) for each one of these models, whereas BOFM generated only 569 features (i.e., active features). As listed in Section II, the feature space size (or number of signatures) is one of the key characteristics determining the practicality and scalability of a malware detector. In BOFM, we achieve this by limiting our total number of features to be a constant number *N*, whereas in [5], the feature space is not constant and grows proportionally to the size of execution traces.

To get a better insight into the feature space problem in Canali et al. [5], let us assume an execution trace with just 100 unique actions with arguments, for example, there are 12 unique actions with arguments in the malware execution trace shown in Figure 1. The simplest malware behavior model: *4-bags of actions with arguments*, generates a feature space of size $\binom{100}{4} = 3{,}921{,}225$. This is very large and the number of features heavily depends on the unique actions present in every single execution trace. This problem is even more acute for other models, such as *2-tuples of actions with arguments* and *2-bags of 2-tuples of actions with arguments*, presented in [5].

In addition, approaches in [5] have high memory requirements and long execution times to perform signature matching, whereas we were able to run standard machine learning algorithms on a standard PC in less than a minute. To be more specific, in [5], it took almost 48 hours to extract malware features using tuples of system calls with arguments, whereas, using BOFM, we were able to extract features in 1.67 hours from the same set of malware samples. Note that our experiments were conducted using a single 4-core Xeon (R) machine with 4GB of RAM, in contrast to [5] where the authors used two clusters: one with eight 4-core Xeon (R) machines with 16GB of RAM and a second one with eight 16-core Authentic AMD machines with 45GB of RAM. Further, we were able to train the SVM classifier, using our training-set, in 26 seconds (averaged over 5 executions), consuming only 200 MB of physical memory (average memory space). In contrast, Canali et al. [5] reported that their prototype malware detector was unable to run on a standard machine due to the huge feature space, as it consumed 1GB of RAM to perform matching on 5 million signatures.

This suggests that BOFM, beyond improvements in accuracy, is also much more efficient and scalable. Further, our approach does not require any complex parameter tuning or preprocessing. From all the above, we can therefore conclude that BOFM, when combined with machine learning algorithms, is indeed a more practical solution than the behavior modeling approaches reported in [5] and other related works, as discussed in Section II.

IX. Conclusion

This paper proposes a novel malware detection solution that combines a new malware behavior modeling technique (BOFM) and machine learning, in order to distinguish malware from benign programs. Our goal is to be sufficiently efficient, scalable, and accurate to complement traditional anti-virus software on end host machines. Our detection models cannot be easily evaded by simple obfuscation techniques as we characterize the behavior of malware as a set of high-level actions that models the interaction between malware and the operating system resources in a systematic manner. Further, the feature space generated by BOFM is of fixed dimension and does not grow in proportion with the number of malware samples under examination. This makes BOFM more efficient and scalable in practice. In addition, in spite of all these practical advantages, when combining BOFM and Support Vector Machines, a well-known machine learning approach, we obtain a better detection accuracy—including no false positives—than reported malware detection techniques. But more importantly, given the usual difficulties in comparing accuracy across studies, such results are obtained with vastly lower computation times and memory usage, thus demonstrating the improved scalability and efficiency of our approach.

Acknowledgements

We would also like to thank David Canali [5] for providing us the malware datasets. Lionel Briand was supported by a PEARL grant from the Fonds National de la Recherche, Luxembourg (FNR).

REFERENCES

[1] Willems, C., Holz, T., & Freiling, F. (2007). Toward automated dynamic malware analysis using CWSandbox. Security and Privacy (S&P), IEEE, 5(2), pp. 32-39.

[2] Bayer, U., Kruegel, C., & Kirda, E. (2006, April). TTAnalyze: A tool for analyzing malware. In 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR).

[3] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., & Kirda, E. (2009, February). Scalable, behavior-based malware clustering. In Network and Distributed System Security Symposium (NDSS).

[4] Christodorescu, M., and Jha, S. (2004). Testing malware detectors. ACM SIGSOFT Software Engineering Notes, 29(4), pp 34-44.

[5] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., & Kirda, E. (2012, July). A quantitative study of accuracy in system call-based malware detection. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA). pp. 122-132. ACM.

[6] You, I., & Yim, K. (2010, November). Malware obfuscation techniques: A brief survey. In International Conference on Broadband, Wireless Computing, Communication and Applications. pp. 297-300.

[7] Kwon, T., & Su, Z. (2011, December). Modeling high-level behavior patterns for precise similarity analysis of software. In Data Mining (ICDM), 2011 IEEE 11th International Conference on (pp. 1134-1139). IEEE.

[8] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., & Wang, X. (2009, August). Effective and efficient malware detection at the end host. In Proceedings of the 18th conference on USENIX security symposium. pp. 351-366. USENIX Association.

[9] Chandramohan, M., Tan, H. B. K., & Shar, L. K. (2012, November). Scalable malware clustering through coarse-grained behavior modeling. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE). ACM.

[10] Russinovich, M. E., & Solomon, D. A. (2004). Microsoft Windows Internals, 4th Edition. Microsoft Windows Server TM 2003, Windows XP, and Windows 2000 (Pro-Developer).

[11] Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. Journal of Computer Security, 19(4), 639-668.

[12] Cuckoo Sandbox: http://www.cuckoosandbox.org/

[13] Christodorescu, M., Jha, S., & Kruegel, K., 2007. Mining specifications of malicious behavior. In Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE). ACM, New York, NY, USA, 5-14.

[14] Egele, M., Kruegel, C., Kirda, E., Yin, H., & Song, D. (2007, June). Dynamic spyware analysis. In Usenix Annual Technical Conference.

[15] Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., & Mitchell, J. (2008). A layered architecture for detecting malicious behaviors. In Recent Advances in Intrusion Detection (RAID). pp. 78-97. Springer Berlin/Heidelberg.

[16] Stinson, E., & Mitchell, J. (2007). Characterizing bots' remote control behavior. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). pp. 89-108.

[17] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter, 11(1), 10-18.

[18] Menard, S. Applied logistic regression analysis (Vol. 106). Sage Publications, Incorporated. (2001).

[19] Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., & Kirda, E. (2010, October). AccessMiner: using system-centric models for malware protection. In Proceedings of the 17th ACM conference on Computer and communications security (CCS). pp. 399-412. ACM.

[20] Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., & Yan, X. (2010, May). Synthesizing near-optimal malware specifications from suspicious behaviors. In Security and Privacy, 2010 IEEE Symposium on (S&P). pp. 45-60. IEEE.

[21] Kirda, E., Kruegel, C., Banks, G., Vigna, G., & Kemmerer, R. (2006, August). Behavior-based spyware detection. In Usenix Security Symposium (Vol. 15).

[22] Moser, A., Kruegel, C., and Kirda, E. (2007, December). Limits of static analysis for malware detection. In Computer Security Applications Conference (ACSAC), Twenty-Third Annual. pp. 421-430. IEEE.

[23] Steinwart, I; and Christmann, A; Support Vector Machines, Springer-Verlag, New York, 2008.

[24] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). Andromaly: a behavioral malware detection framework for Android devices. Journal of Intelligent Information Systems, pp. 1-30.

[25] Symantec Internet Security Threat Report: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf

[26] Jang, J., Brumley, D., and Venkataraman, S. (2011, October). Bitshred: feature hashing malware for scalable triage and semantic analysis. In Proceedings of the 18th ACM conference on Computer and communications security (CCS). pp. 309-320. ACM.

[27] Process and Threads: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684841%28v=vs.85%29.aspx

[28] Nebbett, G. Windows NT/2000 Native API Reference, Macmillan Technical Publishing (MTP), February 15, 2000.

[29] Registry: http://msdn.microsoft.com/en-us/library/windows/desktop/ms724878%28v=vs.85%29.aspx

[30] Carrera, E., & Flake, H. (2008). Automated structural classification of malware. In Proceedings of the RSA Conference. pp. 7-11.

[31] Synchronization: http://msdn.microsoft.com/en-us/library/windows/desktop/ms681924%28v=vs.85%29.aspx

[32] Section Objects and Views: http://msdn.microsoft.com/en-us/library/windows/hardware/ff563684%28v=vs.85%29.aspx