J Comput Virol Hack Tech (2018) 14:181–193
https://doi.org/10.1007/s11416-017-0304-8

Multi-context features for detecting malicious programs

Moustafa Saleh[1] · **Tao Li**[2] · **Shouhuai Xu**[3]

Received: 14 November 2016 / Accepted: 3 August 2017 / Published online: 24 August 2017
© Springer-Verlag France SAS 2017

Abstract Malware detection is still an open problem. There are numerous attacks that take place every day where malware is used to steal private information, disrupt services, or sabotage industrial systems. In this paper, we combine three kinds of contextual information, namely static, dynamic, and instruction-based, for malware detection. This leads to the definition of more than thirty thousand features, which is a large features set that covers a wide range of a sample characteristics. Through experiments with one million files, we show that this features set leads to machine learning based models that can detect both malware seen roughly at the time when the models are built, and malware first seen even months after the models were built (i.e., the detection models remain effective months ahead of time). This may be due to the comprehensiveness of the features set.

Keywords Malware Detection · Machine Learning · Code Obfuscation

✉ Moustafa Saleh
mosale@microsoft.com

Tao Li
taoli@cs.fiu.edu

Shouhuai Xu
shxu@cs.utsa.edu

1 Microsoft Malware Protection Center, Microsoft, One Microsoft Way, Redmond, WA, USA

2 School of Computer Science, Florida International University, Miami, FL, USA

3 Department of Computer Science, University of Texas at San Antonio, San Antonio, TX, USA

1 Introduction

Malicious software, referred to as malware, is a program designed to do harm to individuals, corporations, or governments. Each year a massive amount of malware is produced and propagated worldwide, vendors struggle to keep up with the high number by producing more signatures and maintaining timely updates to their customers. In the 4th quarter of 2014, McAfee received more than 50 million new malicious samples. That means there are 387 new threats every minute [15].

Malware is used in a wide range of attacks, from propagating through the network, launching Denial of Service (DoS) attacks against web servers, and stealing credit card information to more sophisticated attacks against nuclear plants [44]. In addition, financial malware programs are increasing in number and have a stronger impact each year. For example, the malware Carbanak was able to steal a record $1 billion from over 100 financial institutions worldwide [14].

Despite the variety of anti-malware solutions ranging from host-based to network-based, malware detection remains an open problem. The primary contestant to solving the issue is that malware attacks continuously change. Malware programs employ several techniques to avoid detection and to evolve rapidly compared to detection solutions. Techniques that evade signature detection exist, such as simple encryption, polymorphism, and advanced metamorphism [25]. Other techniques target behavior analysis via hook detection [16] and virtual machine detection [22], or even use mimicry attacks to deceive behavioral analysis system by showing benign behavior [43].

A common technique for malware detection is *string* signature. Nevertheless, the technique is not efficient against new malware samples and their variants. Other techniques exist which depend on collecting the behavior data of a pro-

gram to determine if it is malicious or not, but these methods suffer from incomplete coverage of software execution paths. Another technique is to gather data from the file header, such as the number of sections, size of each section, entry point location, etc. This is very fast compared to the former two, however, it could produce a high rate of error if the file is obfuscated. The problem of malware detection has been discussed repeatedly in both academia and business settings, but it is evident that more research and advanced solutions are needed to mitigate this serious threat.

In this paper, we move a step beyond the state of the art by making the following contributions. First, we define and introduce a large set of features, including both static and dynamic aspects of malware samples. Specifically, we define more than 32,000 features. To the best of our knowledge, this is the largest feature set that has been defined for malware analysis. Second, we use Random Forest and Naive Bayes machine learning techniques to learn malware detection models based on this large set of features. We conduct an extensive analysis based on 1,000,000 samples, which are divided equally between malicious and clean files. The malware samples are first seen by VirusTotal [40] during the period from December 2014 to April 2015. Experimental results show that we can achieve 96.7% detection accuracy with a false positive rate of 2.1%. Evolution of both the features and malware families across five months were investigated. The evolution of malware features across months is explained, and a description of how that is correlated to malware campaigns is provided.

The rest of the paper is organized as follows: Section 2 summarizes the related literature. Section 3 describes our features set in details. Section 4 discusses our methodology for constructing the feature vector. Section 5 outlines the system architecture, describing its components and the dataset to be used in experiments. Section 6 describes the different experiments, compares and discusses the results and describes the evolution of features and malware families' distribution during the course of five months. Section 7 concludes the paper.

2 Related work

2.1 Malware detection with dynamic features

Some papers introduced techniques that detect malicious activities by monitoring and logging performance-specific registers of the processor, such as [7,13,36], where [36] used the hardware performance information to detect anomalies in execution and hence can detect exploitation of legitimate software. Other publications use instruction traces during execution such as [35] and [11] . Others used text strings displayed to the user and compared their semantic to the behavior of the file to determine if the malware is carrying on a stealth activity [10].

Salehi et al. [26] uses the APIs and their arguments that the sample uses during execution as the feature vector. Other research studies used temporal and spatial information of the API usage [2], that is, similar to [26], both the APIs and their arguments are collected, however, their execution order is considered as well. Similar research done by Miao et al. [18] utilizes API names extracted during sample execution and constructs two layers of information, one that considers API names, while the other layer is more high level and interprets a sequence of API to infer high level behavior. In addition, Tian et al used API sequences collected during runtime and then applies pattern recognition algorithms on the extracted pattern [37].

2.2 Malware detection with static features

Several types of information can be extracted from the file header, such as imported shared libraries, API names, number of sections, etc. These features could be helpful in providing an overview of the file. Many research studies have been done to detect malware based on this information, such as [21,29, 32,38] which used the PE header and structure information to detect packed files. Other PE header features are also used to detect zero-day malware [33,34]. A seminal work of Shultz *et al* [31] uses data mining techniques to detect malicious software. The research uses the list of DLLs used by the binary, the list of DLL function calls made by the binary, and the number of functions exported by each DLL, all extracted from the PE header. In addition, strings extracted from the file were used as features as well as the byte sequence.

Other techniques used byte n-gram to build a classification model [12], which relies on the file contents. These techniques are prone to evasion via obfuscation. Ahmadi et al. [1] classified 21,741 malicious files from 9 different families with a diverse set of static features such as Entropy, imported APIs, strings, sections, instructions, n-gram of hex values of the file contents, etc. The research claimes that it is capable of classifying obfuscated files. However, the experiment is limited in terms of the number of clusters. Ding et al. [8] on the other hand, extracts the instructions from the statically generated control flow graph. After extracting the instructions, the opcode is extracted and then used by the n-gram method.

Another research done by Invincea Labs [30] for malware detection using Neural Networks achieves 95% detection rate with 0.1% false positive when they used static features in an experiment consisting of 350,016 malicious and 81,910 benign files. The features used were imported DLL names, each DLL's imported functions, Entropy, byte distribution and 256 other features of metadata extracted from the header.

In general, although static analysis is still considered prone to evasion if the file is obfuscated/packed, it is much faster and promising if combined efficiently with machine learning.

2.3 Malware detection with hybrid features

There are a number of publications that use both dynamic and static features for malware detection. Yan *et al*. [42] uses PE header information and n-gram of instructions opcode as static features. For dynamic features, they use n-gram of instruction traces opcode plus the list of invoked system calls. The authors exclude packed files from their dataset and considered only those files that were not flagged by PEiD[3] as packed. Ravula *et al*. [23] gathers dynamic features of added/removed/modified registry keys, whether the sample accesses the internet or not, what DLLs and APIs are used during execution and if the sample accesses another directory. For static features, they considered the packer name by using PEiD [3], the programming language used, list of unique strings and URLs embedded in the files. The total number of features with the list of APIs was 141. Santos *et al*. [28] collects the frequency of occurrences of operation codes that were obtained statically. For dynamic analysis, information is gathered about file activities, some protection mechanisms of the sample, whether the sample is persistent, network activities, process manipulation, whether the sample retrieves information about the system, and unhandled exceptions during execution. The number of dynamic features were 63 while the opcode-sequences frequencies led to 144,598 sparse features array that were later reduced to 1,000. Other interesting research is [4], in which the authors used the raw byte sequence of the binary file, sequence of instruction traces, sequence of the statically disassembled instructions, the control flow graph, and dynamic system call traces. The authors used 1,556 files (780 malicious and 776 benign) as the training set, whereas the test set only contains 20,936 of malicious files.

3 Features definition

We collect an extensive set of the malicious sample features and integrate them to build a feature vector for machine learning techniques. The features are collected from different aspects of the malicious samples. As every aspect of malware can be prone to obfuscation, data collected based on one sole aspect could be inaccurate. This is important because it is intuitive that the more comprehensive the feature set is, the more difficult for the attacker to "obfuscate" the large number of features for evasion. Therefore, we collect several features from many aspects of a malicious file, and use them to build our feature model for describing a system of malware. These feature sets include dynamic analysis of the file when it runs in a controlled environment, static analysis of the file header features, and data based on the instructions that compose the executable file.

Combining these three sets culminates in a feature vector containing all the features. Our vector consists of 32,069 features. In the following subsections, we describe features extracted from each context.

3.1 Instructions features

Instruction features have been successfully used to determined whether a file is obfuscated or not [24]. This inspires us to define instruction features based on the control flow graph and distribution of instructions. The obfuscation feature (i.e., whether a file is obfuscated or not) is presented as one bit in the feature vector.

Using the technique presented in [24], information about the control flow graph of the file is obtained. Based on the control flow graph features, it is determined whether the file is obfuscated, including whether the file is using anti-analysis tricks on the instruction level. A boolean flag is used in the feature vector to indicate whether the file employs any obfuscation techniques. The flag is named *"Is Obfuscated"* in Table 1.

3.2 Static features

Static features consist of both fixed and variable length features. There are 16 fixed length static features, such as the number of file sections, the number of imported APIs, and whether the file is signed or not. There are 3 variable length features, the names of sections, the list of imported APIs, and the list of imported modules names. Three arrays of variable length features account for 4,500 different values, representing features existing across samples in a certain month. For example, the array of imported module names contains 1,500 entries, which are different names found across 200,000 samples gathered from December 2014. Each value is represented as a bit in the feature vector. The length of the static feature vector for the month of December 2014 is 4,516.

It is worth noting that variable-length features differ in length in each month's data. This is because malware samples from a particular month can exhibit different features than those samples collected in a different month. For example, the set of section names of samples collected in December can be different from section names appearing in samples from January.

The file structure can be indicative of the file type, and some research studies are based solely on file structure for malware detection, as mentioned in Sect. 2. Many features from the file header are compiled and used in our final feature vector.

Table 1 Fixed length extracted features

N	Feature	Reported By
1-	Number of imported modules	Anatomist
2-	Number of imported APIs	Anatomist
3-	Entry point	Anatomist
4-	Machine	Anatomist
5-	Image Base	Anatomist
6-	Compile Timestamp	Anatomist
7-	Size Of Headers	Anatomist
8-	Section Alignment	Anatomist
9-	File Alignment	Anatomist
10-	Number of Data Directories	Anatomist
11-	File Size	Anatomist
12-	Number of Sections	Anatomist
13-	Is Suspicious	Anatomist
14-	Is Obfuscated	Anatomist
15-	Link Date	VirusTotal
16-	Is Signed	VirusTotal
17-	Has Hooks	VirusTotal
18-	Has UDP	VirusTotal
19-	Has HTTP	VirusTotal
20-	Has DNS	VirusTotal
21-	Has TCP	VirusTotal
22-	Does Service actions	VirusTotal
23-	Does Process Termination	VirusTotal
24-	Does Process Injection	VirusTotal
25-	Does Process Creation	VirusTotal
26-	Does Modify hosts file	VirusTotal
27-	Does Window Search	VirusTotal
28-	Has Runtime DLLs	VirusTotal
29-	Does Open Mutex	VirusTotal
30-	Does Create Mutex	VirusTotal
31-	Does Delete Registry	VirusTotal
32-	Does Set Registry	VirusTotal
33-	Does Open a file	VirusTotal
34-	Does Move a file	VirusTotal
35-	Does Download a file	VirusTotal
36-	Does Replace a file	VirusTotal
37-	Does Delete a file	VirusTotal
38-	Does Copy a file	VirusTotal

- Imported Modules and API: Many research studies are based on collecting imported API names, as mentioned in Sect. 2. The number of modules and the number of APIs are gathered as two fields in our feature vector. In addition, the list of imported module names and APIs names are collected.
- File Sections: Executable files contain a number of sections for different types of content. Usually *.text* sections contain executable code, *.data* sections contain data, *.rsrc* for storing a program's resources, etc. Some crafted files, usually malicious, can contain unusual section names. In addition, packed files might also have a smaller number of sections than unpacked files, and have specific sections names. For example, UPX packer [39] almost always utilizes sections with the names "UPX0", "UPX1", etc. From every file in our set, section names, file alignment [19] and section alignment [19] are collected.
- EntryPoint: The address of the first instruction to be executed.
- Machine Code: Machine code in the file header. This code indicates the type of the machine or system the file should run on.
- Image Base: Address in memory where the executable file will be located upon execution.
- Compile Timestamp: Timestamp indicating file compilation date.
- Link Date: Timestamp indicating file linking date.
- Size Of Headers: Size of the "optional header" which is, unlike what the name implies, required for PE executable files [19].
- Characteristics: Flags that indicate the attributes of the file, such as if the file is a DLL or not, 32 bit or 64 bit, etc.
- Number of Data Directories: Number of data-directory entries in the optional header.
- File Size: Total file size in bytes.
- Is Suspicious: A value that is set to non-zero if any of the following conditions are met:
  - The entry point is in the file header before any section.
  - There is no .text or CODE section in the file.
  - The entry point is in the last section, which is neither .text nor CODE.
  - SizeOfRawData = 0 and VirtualSize > 0 for some sections.
  - Sum of SizeOfRawData field of all sections is greater than the file size.
  - Two or more sections overlap.
  - The file has no imports at all or the import table is corrupted.
- Digital Signature: Whether the file is signed or not. Although some recent malware samples are signed with stolen private keys, the vast majority of signed software is benign.

3.3 Dynamic features

Dynamic features consist of 22 scalar-value features, such as whether the file utilizes TCP connections, uses API hooking (Yes/No), etc. The variable length features consist of an array of values representing, for example, list of

Hooking types used, HTTP methods, and runtime loaded DLLs. The total number of features represented in the vector of dynamic features for the month of December 2014 is 27,552.

The features extracted from the program's dynamic behavior are as follows:

- Network Activity: A number of features that indicate network activity initiated by the malware sample are extracted. For example, UDP and TCP requests are logged and both the IP and PORT are extracted. Also, DNS requests are used as features, with both the IP and host name collected. In addition, all HTTP requests are collected, and for each request, the User-Agent, Destination URL and Request Method are considered features.
- Services: Service activities are collected, as some malware programs register one or more services to run in the background once they penetrate the system.
- Process Manipulation: Process injection is a highly utilized technique that allows malware to task legitimate processes do the work for them. For example, if the malware needs to have network communication, but the firewall on the system will deny its request, the malware may look for some known processes that will likely have an "allow" rule in the firewall, such as web browsers, and then inject their malicious code to get executed in the context of the legitimate application. A list of injected process names were collected, as well as a number of features related to processes dealt with by the malicious samples during execution, such as whether or not the sample created new processes, process tree names, terminated processes, as well as process injection information. Additionally, a list of shell commands executed by the sample is collected.
- Files: File use is very important for almost every sample. A malicious program could be a virus that infects other executables. In this case, it will look for executable files in the system, open them, and write itself into their code. A trojan could also download other malware programs from the internet. Several types of information about the file's creation, drop, deletion or modification by the sample under analysis are gathered.
- Registry: The system registry is commonly used by processes to save or modify system configurations. Many malicious applications use registry manipulation to achieve their goal. Registry related behavior such as creation, modification and deletion of registry keys or values is noted.
- Mutex Objects: Mutex objects are usually used in multi-threaded applications to control and coordinate shared resources access. Mutexes can have names when created by the application, and the list of the newly created and opened mutexes during the sample runtime are collected.
- Runtime DLL: An executable file can declare the DLLs to be used in the import table, or load the required DLLs during runtime. Usually malware uses the runtime DLL technique to hide behavior. The list of DLLs loaded during runtime is collected and considered as part of the feature vector.
- Windows Manipulation: The list of searched windows names and windows class name are gathered. These values are usually used when the sample calls APIs such as *FindWindow* and *RegisterClass*.
- Extras: The VirusTotal report provides some extra information regarding the use of device drivers, usually whether the file is using the API *DeviceIoControl*, or detecting a debugger presence using *IsDebuggerPresent* API.

4 Feature representation

The feature set consists of two types of features, variable and fixed size. Examples of fixed-length features are *Entry Point*, *Number of Sections*, and *FileAlignment*. The value of each feature in the resulting vector is an integer or boolean value. Table 1 list the fixed features.

The variable size features are the set of features that do not necessarily exist in every file. For example, HTTP host name that the sample connects to, or API and module names imported. Since not all programs have these features, we use an initial set of features that are extracted from a training set of 100,000 malicious samples and 100,000 benign samples. These samples are obtained from VirusTotal. All the variable size features are enumerated from this set and a table is assembled with each different value.

Table 3 shows the top ten modules that are extracted from the 100,000 malicious files in the training set corresponding to December 2014, with their occurrence frequency (i.e., the number of malicious files where a specific module name is found).

Let us use an example to demonstrate how variable-length features are represented. Suppose we use the ten modules mentioned in Table 3 as features. Then, we will have a binary feature vector of size 10, where a vector element with value '1' means that the program uses the corresponding module and '0' otherwise. For instance, if a program uses modules KERNEL32.DLL, LIBC.DLL, SHELL32.DLL, MSVCRT.DLL, and SHLWAPI.DLL, it will have a feature vector of [1, 0, 0, 0, 0, 0, 1, 1, 0, 1]. In case of module LIBC.DLL, it was used by the file, but it is not in our base features, so it will not be a part of the vector. The same process is done for the other features. As such, each feature in the resulting vector is a boolean value. Table 2 lists the variable-length features.

Table 2 Variable Length Extracted Features

N	Feature	Reported By
1-	Modules	Anatomist
2-	APIs	Anatomist
3-	Section Names	Anatomist
4-	Hook Type	VirusTotal
5-	Hook Method	VirusTotal
6-	HTTP URL	VirusTotal
7-	HTTP Method	VirusTotal
8-	HTTP User Agent	VirusTotal
9-	DNS IP	VirusTotal
10-	DNS hostname	VirusTotal
11-	UDP IP:PORT	VirusTotal
12-	TCP IP:PORT	VirusTotal
13-	Opened services names	VirusTotal
14-	Opened services managers database	VirusTotal
15-	Opened services managers machine	VirusTotal
16-	Extra	VirusTotal
17-	Shell cmd	VirusTotal
18-	Process Tree Name	VirusTotal
19-	Injected Process Name	VirusTotal
20-	Created Processes Name	VirusTotal
21-	Terminated Processes Name	VirusTotal
22-	Searched Windows Class	VirusTotal
23-	Searched Windows Name	VirusTotal
24-	Runtime DLLs	VirusTotal
25-	Opened Mutex Name	VirusTotal
26-	Created Mutex Name	VirusTotal
27-	Set Registry Entry Type	VirusTotal
28-	Set Registry Entry Key	VirusTotal
29-	Set Registry Entry Value	VirusTotal
30-	Deleted Registry Entry Key	VirusTotal
31-	Opened Files path	VirusTotal
32-	Read Files path	VirusTotal
33-	Written Files path	VirusTotal
34-	Deleted Files path	VirusTotal

Table 3 Top Ten Module Names in 100k malicious files

	Module Name	Number of occurrences
1	KERNEL32.DLL	85,046
2	USER32.DLL	69,724
3	ADVAPI32.DLL	58,098
4	OLEAUT32.DLL	37,622
5	OLE32.DLL	34,280
6	GDI32.DLL	34,092
7	SHELL32.DLL	31,664
8	MSVCRT.DLL	24,642
9	COMCTL32.DLL	20,677
10	SHLWAPI.DLL	17,412

5 System design

In this paper, we focus on Windows PE executable files. We choose one million files for our experiment, which is, to the best of our knowledge, the largest number of files used in academic literatures of malware detection. We exclude Adware from consideration because Adware programs, which display unwanted ads without user consent or permission, have been used by legitimate companies [6]. This suggests that detecting Adware programs may need a different approach than detecting malicious programs. We obtain both malicious and benign binary programs, along with the scan report of each program from VirusTotal. The malicious files in our experiment are defined as those files that were flagged by 20 or more antivirus programs, including Microsoft, Kaspersky, McAfee and Bitdefender anti-malware. Since Kaspersky, McAfee and Bitdefender are among the best tools in regard to detection rate and Microsoft is the best in regard to false-positive rates [5], we assure that all malicious files in our experiment were detected by all four anti-malware programs. On the other hand, benign programs are those which are not flagged as malicious by any of the 56 antivirus systems at VirusTotal. We understand that even though each file is examined by numerous antivirus systems and found to be clean, there is still a possibility that the file is misclassified. However, automated scanning with a multitude of antivirus systems appears to be the only feasible way to cope with the large set of samples we study.

The programs were first seen by VirusTotal during the period from December 2014 to April 2015. We collected 100,000 malicious programs and 100,000 benign programs during each month between December 2014 and April 2015, which leads to a total of 1,000,000 programs.

The scan report from VirusTotal is a JSON file containing static and dynamic analysis data of the file. The static analysis data is obtained using the `pefile` tool [20], while the dynamic analysis is done by a modified version of Cuckoo [27]. VirusTotal lets a program run for up to 60 seconds before it is terminated [9,17].

We utilized a C++ PE file scanner called Anatomist. It extracts different types of information from the header of a PE file, as well as its control flow graph and a list of instructions. We adopt this third-party tool [24] because it is resilient to many tricks that have been used by malware developers to hinder the process of extracting header information. Anatomist outputs a report in JSON format. Both Anatomist and VirusTotal reports are stored in a MySQL database.

Figure 1 highlights our system. It works as follows: We download the binary files with their reports. Then each binary

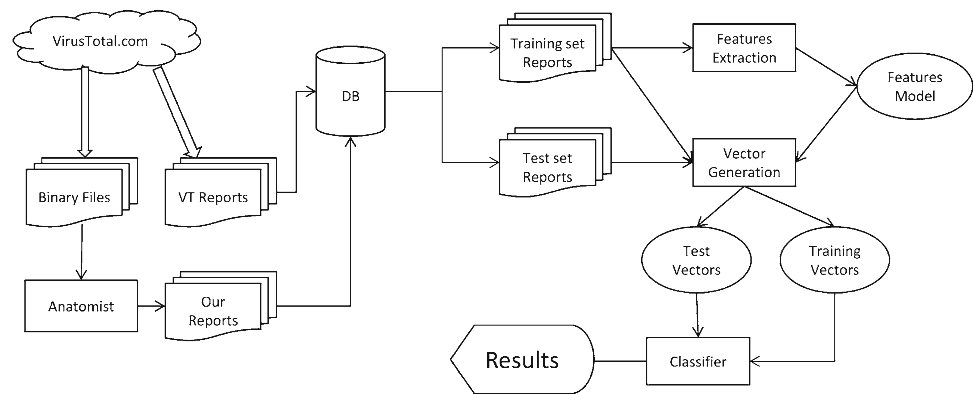

Fig. 1 Our system architecture

file is fed to Anatomist. To train classification models, the training set of reports of both the clean and malicious programs are retrieved from the database. Then, features are extracted from the reports and represented as a key/value map in the `Features Model`. The key represents the feature name, while the value represents the number of occurrences of that feature in the training set. This key/value map is similar to Table 3, for each variable-length feature.

Both the training set and the test set are fed to the `Vector Generation` component which will match each report to the `Features Model` and get a feature vector for each program according to the feature representation discussed in Section 4. Thus, `Vector Generation` will generate a list of training and test set vectors. The generated vectors will be the input to the machine learning classifier. We use WEKA [41] as our machine learning tool.

6 Experiment and results

For every month i, we have a set of files, denoted by S_i. For an experiment, we use training set S_i and test set is $\frac{1}{2}S_j$ where $i \neq j$, meaning that the experiment use 66% data from one month for training and 33% data from another month for testing. We denote such an experiment as $E(S_i, \frac{1}{2}S_j)$. In the experiment, we have 5 months of data, namely $i, j \in$ {Dec14, Jan15, Feb15, Mar15, Apr15}, where "Dec14" means December 2014 and so on.

Practically speaking, we have a *training set* from which we extract the variable length base features. The training set is 200,000 files distributed equally between clean and malicious files. We create a feature vector for each file to train the model. After that we use a test set of 100,000 files distributed equally between clean and malicious files, where we match their properties against the features extracted from the training set, and form a feature vector for each file. So in total we have a training set of 200,000 files and a test set of 100,000 files from next month both distributed equally between clean and malicious files.

Table 4 Top Ten Modules with Highest Difference of Occurrence between Clean and Malicious Files

ID	Module Name	# of Clean files	# of Malicious files
1	COMCTL32.DLL	56,615	20,677
2	SHELL32.DLL	53,521	31,664
3	VERSION.DLL	35,379	14,044
4	ADVAPI32.DLL	78,972	58,098
5	OLE32.DLL	54,298	34,280
6	GDI32.DLL	53,667	34,092
7	MSVCRT.DLL	8,559	24,642
8	OLEAUT32.DLL	51,389	37,622
9	USER32.DLL	81,312	69,724
10	KERNEL32.DLL	96,034	85,046

As an example of our base features, Table 4 shows the top ten modules that have the highest difference of occurrence in training files from December 2014. For example, the module "COMCTL32.DLL" is used by 56,615 clean files and 20,677 malicious ones, with difference of 35,938.

We use the following standard metrics for measuring the detection effectiveness. Let TP (True Positive) be the number of malicious files that were correctly classified as malicious, FP (False Positive) be the number of benign files that were misclassified as malicious, FN (False Negative) be the number of malicious files that were incorrectly classified as benign, and TN (True Negative) be the number of correctly classified benign files. Detection accuracy is defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Another commonly used metric is the area under the receiver operating characteristic (ROC) curve, which is utilized here as a performance evaluation criterion.

We use two popular machine learning algorithms to conduct our experiments: Naive Bayes and Random Forest. In order to understand how the comprehensiveness of features

Table 5 Results of experiments $E(S_i, \frac{1}{2}S_j)$, where i = Dec14 and $j \in$ {Jan15, Feb15, Mar15, Apr15}

		Static Features Only		Dynamic Features Only		Static and Dynamic Features	
		RF	NB	RF	NB	RF	NB
January 2015	TP	95.6%	91.9%	85.1%	42.1%	95.5%	92.1%
	FP	2.8%	40.9%	0.6%	8.5%	2.1%	40.3%
	Accuracy	96.4%	75.5%	92.3%	66.8%	96.7%	75.9%
	ROC	99.3%	77.5%	95.0%	85.9%	99.5%	78.5%
February 2015	TP	89.6%	92.4%	85.8%	66.5%	94.3%	92.7%
	FP	4.8%	37.2%	2.1%	15.2%	2.8%	36.8%
	Accuracy	92.4%	77.6%	91.8%	75.6%	95.7%	78%
	ROC	96.9%	78.8%	96.7%	86.3%	98.7%	80.1%
March 2015	TP	94.0%	92.4%	65.8%	65.1%	94.9%	92.6%
	FP	2.5%	34.1%	1.4%	12.1%	1.5%	33.7%
	Accuracy	95.7%	79.1%	82.2%	76.5%	96.7%	79.5%
	ROC	98.4%	82.0%	94.2%	82.8%	98.8%	82.9%
April 2015	TP	86.3%	90.9%	75.4%	80.1%	88.5%	91.2%
	FP	2.2%	24.4%	1.5%	35.3%	1.2%	24.1%
	Accuracy	92.0%	83.3%	87.0%	72.4%	93.6%	83.5%
	ROC	98.5%	84.1%	96.9%	77.7%	99.4%	84.8%

can impact the effectiveness of malware detection, we consider three scenarios in our experiment:

- experiments using static features only,
- experiments using dynamic features only, and
- experiments using both static and dynamic features.

6.1 Experiments $E(S_i, \frac{1}{2}S_j)$

6.1.1 Static features only

Static features are features 1-16 in Table 1 and features 1-3 in Table 2. Figure 2(a) and Table 5 show the performance of Naive Bayes and Random Forest algorithms, referred to as RF and NB in Table 5, respectively. It can be noted that the Random Forest does a better job than Naive Bayes in general on the four metrics we have (TP, FP, Accuracy and ROC), except for the month of February and April where Naive Bayes achieves higher TP. However, although Naive Bayes achieves 92.4% true positive detection in February, its false positive error of 37.2% is too high to be considered applicable in any production scenario.

6.1.2 Dynamic features only

In this scenario, we consider dynamic features only; these are the features from 17 through 38 in Table 1 and from 4 through 34 in Table 2. According to Figure 2(b) and Table 5 that illustrate the performance of Naive Bayes and Random Forest. It is apparent that Random Forest still outperforms Naive Bayes in dynamic features test. However, it can be noted that the overall accuracy is less than the accuracy achieved by the same classifiers with regard of static features. This is probably because the dynamic features are less rich than the static ones in our dataset. The dynamic analysis box at VirusTotal executes the sample for only one minute on a custom version of Cuckoo sandbox. We believe that this time is insufficient to capture some of the important behavior of the malware, especially insufficient for a sample that needs to communicate over the internet before it shows its malicious behavior. In our future work, we plan to have our own dynamic analysis system to overcome this time limit of analysis.

6.1.3 Combined features

In the final scenario, we consider all features, both static and dynamic. This led to the best classification results. Figure 2(c) and Table 5 show the performance of the Naive Bayes and Random Forest algorithms, while Figure 2(d) shows a comparison of Random Forest performance in the three scenarios. We can tell from the result that Random Forest performed best on every month with both static and dynamic features combined, which highlights the importance of the integration of static and dynamic features.

6.2 Experiments $E(S_i, \frac{1}{2}S_{i+1})$

An additional experiment was conducted to check the result of testing each month's training data on the following month

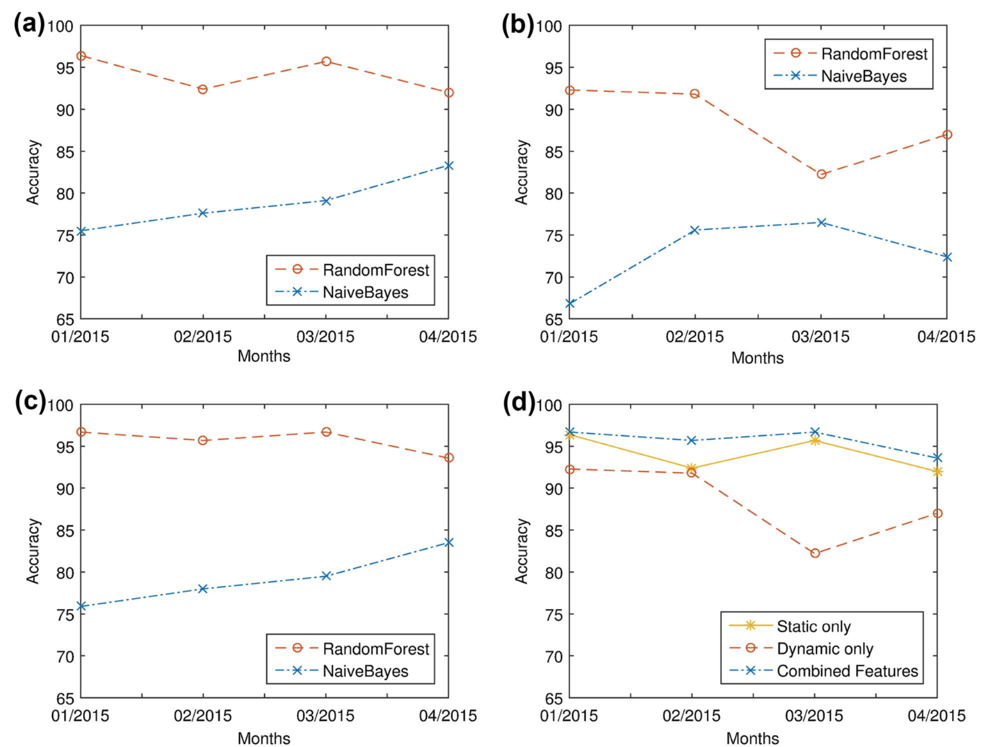

Fig. 2 Results of experiments $E(S_i, \frac{1}{2}S_j)$ where i = Dec14 and $j \in$ {Jan15, Feb15, Mar15, Apr15}. (a) Accuracy using static features only. (b) Accuracy using dynamic features only. (c) Accuracy using static and dynamic features. (d) Accuracy of Random Forest using the three features sets

Table 6 Results of experiment $E(S_i, \frac{1}{2}S_{i+1})$, where $i \in$ {Dec14, Jan15, Feb15, Mar15} and $i + 1$ means the month after i

Classifier	Period	TP	FP	Acc	ROC
Random	Dec14-Jan15	95.5%	2.1%	96.7%	99.5%
Forest	Jan15-Feb15	98%	1.3%	98.3%	99.8%
	Feb15-Mar15	96.3%	1.1%	97.6%	99.7%
	Mar15-Apr15	96.4%	0.9%	97.8%	99.7%
Naive	Dec14-Jan15	92.1%	40.3%	75.9%	78.5%
Bayes	Jan15-Feb15	90.3%	29.1%	80.6%	85.3%
	Feb15-Mar15	96%	42.5%	76.8%	78.2%
	Mar15-Apr15	90.1%	24%	83.1%	83.9%

utilizing all features. Table 6 illustrates the results of the experiment. The experiment showed the different results when using training sets from different months. Again, this experiment emphasizes that Random Forest outperforms Naive Bayes.

6.3 Discussion

Our discussion mainly focuses on Random Forest, because it outperforms Naive Bayes in every experiment. The experiment showed several interesting points worth discussing,

First, it is expected that the accuracy will gradually decrease when predicting future months, since new malware applications could emerge that differ significantly from those in the initial training set. However, as can be noticed in Table 5, some months show a gradual drop in performance, while others show a gain. For example, Random Forest performed well using only dynamic features until 3/2015, which had a sudden drop of performance as shown in Figure 2(b). There are two primary reasons for this; The first reason is the surge of samples from malware families that were under-represented in the training data. While the second reason is the emergence of new malware families that were not represented in the initial training set. For example, the malware family `Win32/Loring` accounts for 2,147 files in December 2014, but the number of files increased in March 2015 to 11,763 files, representing the top family at almost 23.5% of the March 2015 test files. Conversely, new malware families emerged in March, such as `Win32/Rofin`, with 228 files and `Win32/Delf` with 118 files. The total number of new families in March is 265 with 1033 different files. Table 7 shows the top ten malware families and the ratio of each family to the total number of files in each month. Family names as shown in the table are obtained from Microsoft anti-virus labels. The first column in the table represents the family distribution in training data (100,000 malicious files), whereas the other four columns represent distribution in the test data (50,000 files each).

The training data for December 2014 contains 1,259 unique malware families, with 3,049 total variants. Some families have only one variant present in the training data, such as `Win32/Claretore`, while other families contain over a hundred variants, such as `Win32/Vobfus`. A sin-

Table 7 Top Ten Malware Families in Each Month

Training December 2014		Test January 2015		Test February 2015		Test March 2015		Test April 2015	
Families	Count	Families	Count	Families	Count	Families	Count	Families	Count
Virut	16.66%	Eggnog	34.22%	Upatre	14.85%	Loring	23.53%	Virut	17.17%
Vflooder	4.60%	Virut	7.50%	Nabucur	8.34%	Expiro	14.69%	Lydra	9.57%
Elkern	4.55%	Upatre	5.01%	Soltern	7.69%	Vobfus	7.57%	Soltern	8.94%
Parite	4.39%	Simbot	4.11%	Virut	7.47%	Virut	5.95%	Vflooder	7.48%
Jadtre	4.13%	Loring	3.89%	Vobfus	4.31%	Delf	5.59%	Loring	4.23%
Sality	3.26%	Parite	3.74%	Expiro	4.15%	Mydoom	5.14%	Rofin	4.17%
Ramnit	2.44%	Viking	2.96%	Loring	3.96%	Worm:Win32/VB	5.12%	Trojan:Win32/VB	3.64%
Almanahe	2.40%	Beaugrit	2.70%	Sality	3.48%	Vflooder	4.53%	Sality	3.53%
Loring	2.15%	Vflooder	2.67%	Comame!gmb	3.39%	Comame!gmb	3.57%	Worm:Win32/VB	3.30%
Gupboot	1.99%	Sality	2.39%	Berbew	3.33%	Beaugrit	2.32%	Morefi	3.16%

gle variant of a family can appear in one file in the training set, while other variants appears in several files, such as `Win32/Upatre.A` which appears in 707 different files. The top families in December 2014, shown in table 7 and in regard to the total number of files, `Win32/Virut` takes the lead, where the family has a total of 16,666 files. Its two most common variants are `Win32/Virut.BR` and `Win32/Virut.BN` with 6,728 and 4,841 files, respectively.

Second, we observe interesting phenomenon on feature evolution. With a feature vector of length 32,000+ and over a million files, it is computationally difficult to apply feature reduction algorithms to ascertain the importance of individual features. In order to get a sense of features importance, and for the purpose of illustration, we examined the occurrence of features in the benign and malicious files. A feature is rated as important when it can distinguish between each class. For example, as shown previously in Table 4 the DLL `COMCTL32.DLL` was present in clean files more than malicious ones. The difference between the number of occurrence of the DLL in each class was the highest among other DLLs. We consider the disparity of occurrence, rather than the mere number of occurrence, to determine if a feature is more distinguishing than other features.

We selected three classes of features to illustrate how the feature evolution progresses month to month. Table 8 shows section names, evolution of APIs, and injected processes, respectively, observed in the data of each month. Each month contains 200,000 files divided evenly between clean and malicious categories, resulting in one million files of five months.

Each column in Table 8 shows the month of observation, feature name, and number of files observed having that feature. The features listed are the top five distinguishing features in that class. For example, in Table 8 the API `Sleep` observed in 65,573 and 30,414 clean and malicious files, respectively, providing a difference of 35,159 additional files in the *Clean* category. This was the highest difference among APIs, and thus the top distinguishing API in the API features. Likewise, the *Malware* row in the table lists the most distinguishing features, with the highest contrast favoring malicious files.

An interesting observation in Table 8 is the high number of `_controlfp`, `__p__commode` and other C-runtime floating point functions in four months, but not in January 2015. This can be understood when we look at January's malware families in Table 7. In January 2015, there was a surge in the number of files of the `Eggnog` family, utilizing registry APIs extensively, and accounting for 34.22% of files in that month. This made its APIs dominate the top five malware APIs that month.

In `Section Names` category, it is evident from Table 8 that UPX packer is heavily used by malicious files since most distinguishing section names are UPX0, UPX1, etc. UPX is a

Table 8 Top five features in three categories with highest differences of usage between malicious and clean files

	December 2014		January 2015		February 2015		March 2015		April 2015	
	Value	Files	Value	Files	Value	Files	Value	Files	Value	Files
APIs										
Clean	Sleep	65,573	Sleep	65,729	MultiByteToWideChar	69,448	DestroyWindow	60,392	DestroyWindow	65,527
	DestroyWindow	50,603	GetCurrentProcess	66,667	SetFilePointer	63,675	SetFilePointer	64,695	Sleep	78,919
	WriteFile	61,018	MultiByteToWideChar	59,977	GetSystemMetrics	50,013	GetExitCodeProcess	51,651	GetExitCodeProcess	57,957
	ReadFile	54,884	GetLastError	66,898	SetWindowPos	47,909	ReadFile	65,025	GetCurrentProcess	75,535
	SetFilePointer	53,836	FreeLibrary	53,781	WriteFile	68,699	Sleep	74,499	MultiByteToWideChar	66,688
Malware	_controlfp	19,860	RegFlushKey	19,511	_controlfp	21,698	_controlfp	20,745	VirtualProtect	34,450
	__p__commode	21,630	RegSetValueExA	28,365	__p__commode	23,281	__p__commode	24,323	_controlfp	15,408
	__p__fmode	22,112	RegCreateKeyExA	26,161	__p__fmode	23,809	__p__fmode	23,392	__p__commode	16,601
	_except_handler	17,022	RegQueryValueExA	33,492	_except_handler3	18,823	_c_exit	16,114	__p__fmode	16,906
	_c_exit	12,265	RegOpenKeyExA	33,096	exit	28,770	_cexit	23,383	_c_exit	10,331
Section Names										
Clean	.rdata	76,451	.data	81,421	.rdata	86,894	.rdata	80,807	.rdata	85,326
	.reloc	44,030	.rdata	76,326	.reloc	54,451	.data	81,372	.data	82,194
	.idata	28,673	.text	80,127	.idata	38,278	.text	80,992	.text	82,612
	.tls	21,951	.ndata	18,308	.tls	26,122	.ndata	22,246	.ndata	31,978
	.data	72,863	.rsrc	94,772	.ndata	16,994	.reloc	39,069	.rsrc	98,094
Malware	UPX0	26,181	DATA	25,239	UPX0	12,209	UPX0	27,235	UPX0	49,548
	UPX1	26,140	BSS	24,040	UPX1	12,087	UPX1	27,209	UPX1	49,460
	UPX2	5,011	CODE	24,165	rsrc	2,785	.aspack	14,808	UPX2	15,673
	.aspack	3,141	.tls	26,025	uinC	2,777	.adata	15,017	.adata	3,585
	.adata	3,274	.aspack	4,779	.vmp0	2,685	ExeS	13,326	.aspack	3,258
Injected Processes										
Clean	SELF	2,750	msiexec.exe	369	msiexec.exe	351	SELF	2,027	SELF	2,281
	dwwin.exe	2,407	SELF	2,207	tmp1.exe	24	msiexec.exe	935	msiexec.exe	893
	msiexec.exe	717	drwtsn32.exe	70	DTLService.exe	22	dwwin.exe	2,061	GameCenter@Mail.Ru.exe	81
	wmiprvse.exe	325	DTLService.exe	14	LMIGuardianSvc.exe	19	HssInstaller.exe	96	drwtsn32.exe	97
	MiniThunderPlatform.exe	81	TenioDL.exe	12	LMI_Rescue_srv.exe	10	af_proxy_cmd_rep.exe	66	wmic.exe	55
Malware	explorer.exe	1,228	explorer.exe	1,616	SELF	6,904	explorer.exe	1,679	explorer.exe	1,322
	python.exe	763	iexplorer.exe	805	iexplorer.exe	1,829	services.exe	1,490	services.exe	898
	services.exe	658	cmd.exe	746	explorer.exe	1,746	cmd.exe	853	dwwin.exe	2,593
	cmd.exe	627	net.exe	703	python.exe	1,233	net.exe	711	python.exe	629
	poskAAUk.exe	547	pAAMUcMg.exe	678	biudfw.exe	1,026	com7.exe	611	iexplorer.exe	527

free and an easy to use packer, and although it can be used in clean files for packing, it is frequently abused by adversaries to obfuscate executables and evade string signature. Behind UPX is "ASPack" packer, which is a commercial packer and also used for similar obfuscation.

Code injection is commonly used by malware to hide malicious code inside trusted processes. However, process code injection is still used by legitimate software. Table 8 illustrates the injected process names used by both benign and malicious files. It can be observed that "SELF" is the most injected process with respect to the number of clean files. SELF describes the same running process. This is done usually when a plugin or a piece of code is received and loaded by the running process, then injected into the current context as a thread. On the other hand, it is obvious from the table that "explorer.exe" is the mostly injected process by malware. This behavior was illustrated by `Win32/Sality` and `Win32/Madang`. As a side note, for the file "GameCenter@Mail.Ru.exe", obviously the file has an interesting name that could give the impression that it is malicious. However, we analyzed the file and it was found to be clean and legitimately signed.[1]

7 Conclusion

Extensive analysis is presented and applied to a diverse set of one million files, both malicious and clean, where thousands of features were extracted. We defined and integrated features from three contexts to describe a malicious file and used Random Forest and Naive Bayes machine learning algorithms to create and train classifiers to detect malicious programs. In one experiment, a classifier was trained on a single month's data and used to detect malware from future months, up to four months ahead with a high detection accuracy up to 96.7% and false positive rate of 2.1%. The evolution of features and malware families over the course of five months were also studied and illustrated. As future work, we aim to enhance our dynamic analysis data by building an in-house system for dynamic analysis that executes the sample for more than one minute. This will give us a richer dynamic data and will have a positive impact on our detection rate. We also plan to apply feature selection algorithms to get the most important features. In addition, we plan to leverage the system capabilities to cluster the samples into families as well as consider other file types.

Acknowledgements We thank VirusTotal for providing us the dataset that is analyzed in the present paper. We also thank John Charlton for proofreading the paper. The research was supported in part by ARO Grant #W911NF-13-1-0141, NSF Grants #1111925, #IIS-1213026 and #CNS-1461926.

[1] For more information, see https://goo.gl/FCEPLh

References

1. Ahmadi, M., Giacinto, G., Ulyanov, D., Semenov, S., Trofimov, M.: Novel feature extraction, selection and fusion for effective malware family classification. ArXiv e-prints (2015)
2. Ahmed, F., Hameed, H., Shafiq, M.Z., Farooq, M.: Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In: Proceedings of the 2Nd ACM Workshop on Security and Artificial Intelligence, AISec '09, pp. 55–62. ACM, New York, NY, USA (2009). doi:10.1145/1654988.1655003
3. aldeid.com: PEiD. http://www.aldeid.com/wiki/PEiD. Accessed: Feb. 8th, 2014
4. Anderson, B., Storlie, C., Lane, T.: Improving malware classification: bridging the static/dynamic gap. In: Proceedings of the 5th ACM workshop on Security and artificial intelligence, pp. 3–14. ACM (2012)
5. AV-Comparative: File detection test of malicious software. (March 2015)
6. CNET: lenovo hit by lawsuit over superfish adware. http://www.cnet.com/news/lenovo-hit-by-lawsuit-over-superfish-adware/. Accessed 9 December 2015
7. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S.: On the feasibility of online malware detection with performance counters. SIGARCH Comput. Archit. News **41**(3), 559–570 (2013). doi:10.1145/2508148.2485970
8. Ding, Y., Dai, W., Yan, S., Zhang, Y.: Control flow-based opcode behavior analysis for malware detection. Computers & Security **44**, 65–74 (2014). doi:10.1016/j.cose.2014.04.003. http://www.sciencedirect.com/science/article/pii/S0167404814000558
9. Hiramoto, K.: Technical account manager at VirusTotal. Personal Communication. Sept. 24th, 2014
10. Huang, J., Zhang, X., Tan, L., Wang, P., Liang, B.: Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 1036–1046. ACM, New York, NY, USA (2014). doi:10.1145/2568225.2568301
11. Kang, B., Han, K.S., Kang, B., Im, E.G.: Malware categorization using dynamic mnemonic frequency analysis with redundancy filtering. Digit. Investig. **11**(4), 323–335 (2014). doi:10.1016/j.diin.2014.06.003. http://www.sciencedirect.com/science/article/pii/S1742287614000772
12. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, pp. 470–478. ACM, New York, NY, USA (2004). doi:10.1145/1014052.1014105
13. Kompalli, S.: Using existing hardware services for malware detection. In: Security and Privacy Workshops (SPW), 2014 IEEE, pp. 204–208. IEEE (2014)
14. Labs, K.: The great bank robbery: the carbanak apt. http://securelist.com/blog/research/68732/the-great-bank-robbery-the-carbanak-apt/. Accessed 25 Mar 2015
15. Labs, M.: Mcafee labs threats report for february 2015. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2014.pdf. Accessed 25 Mar 2015
16. M0SA: Syp.01: Bypassing online dynamic analysis systems. Valhalla ezine, issue #4, November 2013. http://vxheaven.org/lib/vmo04.html
17. Martinez, E.: Software engineer at VirusTotal. Personal Communication. Dec. 25th, 2014
18. Miao, Q., Liu, J., Cao, Y., Song, J.: Malware detection using bilayer behavior abstraction and improved one-class support vector machines. Int. J. Inf. Secur. **15**(14), 1–19 (2015). doi:10.1007/s10207-015-0297-6

19. Microsoft: Microsoft pe and coff specification. https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx. Accessed 20 Nov 2015
20. pefile: https://github.com/erocarrera/pefile. Accessed 6 June 2015
21. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recogn. Lett. **29**(14), 1941–1946 (2008). doi:10.1016/j.patrec.2008.06.016
22. Quist, D., Smith, V., Computing, O.: Detecting the presence of virtual machines using the local data table. Offens. Comput. (2006)
23. Ravula, R.R., Liszka, K.J., Chan, C.C.: Learning attack features from static and dynamic analysis of malware. In: Knowledge Discovery, Knowledge Engineering and Knowledge Management, pp. 109–125. Springer (2013)
24. Saleh, M., Ratazzi, E., Xu, S.: Instructions-based detection of sophisticated obfuscation and packing. In: Military Communications Conference (MILCOM), 2014 IEEE, pp. 1–6 (2014). doi:10.1109/MILCOM.2014.9
25. Saleh, M.E., Mohamed, A.B., Nabi, A.A.: Eigenviruses for metamorphic virus recognition. IET Inf. Secur. **5**(4), 191–198 (2011)
26. Salehi, Z., Sami, A., Ghiasi, M.: Using feature generation from API calls for malware detection. Comput. Fraud Secur. **2014**(9), 9–18 (2014)
27. Sandbox, C.: Cuckoo sandbox: automated malware analysis. Accessed 6 June 2015
28. Santos, I., Devesa, J., Brezo, F., Nieves, J., Bringas, P.G.: Opem: a static-dynamic approach for machine-learning-based malware detection. In: International Joint Conference CISIS12-ICEUTE' 12-SOCO' 12 Special Sessions, pp. 271–280. Springer (2013)
29. Santos, I., Ugarte-Pedrero, X., Sanz, B., Laorden, C., Bringas, P.G.: Collective classification for packed executable identification. In: Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference, CEAS '11, pp. 23–30. ACM, New York, NY, USA (2011). doi:10.1145/2030376.2030379
30. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. arXiv preprint arXiv:1508.03096 (2015)
31. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Proceedings 2001 IEEE Symposium on Security and Privacy, 2001. S&P 2001, pp. 38–49. IEEE (2001)
32. Shafiq, M., Tabish, S., Farooq, M.: PE-probe: leveraging packer detection and structural information to detect malicious portable executables. In: Proceedings of the Virus Bulletin Conference (VB), pp. 29–33 (2009)
33. Shafiq, M., Tabish, S., Mirza, F., Farooq, M.: PE-Miner: Mining structural information to detect malicious executables in real-time. In: E. Kirda, S. Jha, D. Balzarotti (eds.) Recent Advances in Intrusion Detection. Lecture Notes in Computer Science, vol. 5758, pp. 121–141. Springer, Berlin Heidelberg (2009). doi:10.1007/978-3-642-04342-0_7
34. Shahzad, F., Farooq, M.: Elf-miner: using structural knowledge and data mining methods to detect new (linux) malicious executables. Knowl. Inf. Syst. **30**(3), 589–612 (2012). doi:10.1007/s10115-011-0393-5
35. Storlie, C., Anderson, B., Vander Wiel, S., Quist, D., Hash, C., Brown, N.: Stochastic identification of malware with dynamic traces. ArXiv e-prints (2014)
36. Tang, A., Sethumadhavan, S., Stolfo, S.J.: Unsupervised anomaly-based malware detection using hardware features. CoRR arXiv:1403.1631 (2014)
37. Tian, R., Islam, M., Batten, L., Versteeg, S.: Differentiating malware from cleanware using behavioural analysis. In: 2010 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 23–30 (2010). doi:10.1109/MALWARE.2010.5665796
38. Treadwell, S., Zhou, M.: A heuristic approach for detection of obfuscated malware. In: IEEE International Conference on Intelligence and Security Informatics, 2009 ISI '09, pp. 291–299 (2009). doi:10.1109/ISI.2009.5137328
39. UPX: Upx: The ultimate packer for executables. http://upx.sourceforge.net/. Accessed 7 Dec 2015
40. VirusTotal: http://www.VirusTotal.com/. Accessed 6 June 2015
41. Weka: Weka 3: Data mining software in Java. http://www.cs.waikato.ac.nz/ml/weka/. Accessed 6 June 2015
42. Yan, G., Brown, N., Kong, D.: Exploring discriminatory features for automated malware classification. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 41–61. Springer (2013)
43. You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: BWCCA, pp. 297–300 (2010)
44. Zetter, K.: Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon. Crown Publishing Group, New York (2014)