Engineering Applications of Artificial Intelligence 59 (2017) 93–102

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values

Zahra Salehi, Ashkan Sami*, Mahboobe Ghiasi

CSE & IT Department, Shiraz University, Shiraz, Iran

ARTICLE INFO

Keywords:
Dynamic malware analysis
Behavior malware analysis
Malware detection
Return value API calls arguments
Feature generation

ABSTRACT

Basically malware detection techniques are either: static analysis or dynamic analysis. Static analysis explores malware code without executing it while dynamic analysis relies on run-time values. Static analysis suffers from obfuscation but dynamic analysis is less sensitive to code obfuscation. In this paper, a new dynamic malware feature selection method is proposed that mainly is based on novel feature generation. Similar to other dynamic methods, each binary is run in a controlled environment. The arguments and return values of each respective API call are recorded. Features are constructed based on the name of API calls and each argument and/or return value recorded during runtime. A selected set of features have such a discriminative capability that can be used to classify with an accuracy of 99.4% and a false positive rate less than one percent on a 1211 malware and benign PEs dataset. Features are so robust that even on much larger datasets containing new families of malware accuracy of 96.3% on a 3175 new samples with the selected features of the first experiment is obtained. This setting proves the features can present malicious activity irrespective of dataset families. List of executables, source code and execution traces can be found at: http://home.shirazu.ac.ir/~sami/malware

1. Introduction

Malware is software that designed to infect, infiltrate or intrude a computer system without the owner's authentication. Malware comes from two words: "malicious" and "software". Unfortunately new malware types are emerging annually. Symantec (2011) reported an increase of more than 81% from 2010 in malware attacks. According to McAfee (2015), more than 400 million types of malware have been detected during the second half of 2015. Nowadays a new family of malwares has derived from highly engineered pieces of software that are able to carry out advanced, large-scale attacks (Sood and Enbody, 2013). High-profile attacks such as Stuxnet, discovered in 2010 targets critical industrial infrastructure (Hu, 2011), DuQu (2011), Flame (2012), Gauss (2014) and the rise of Anonymous-centric hacktivism made recent years a truly challenging for the security professionals.

Wide spread use of computer networks facilitates the growth of malware. Connectivity increases the visibility of vulnerable systems and paves the path to the exploitations of these vulnerabilities. Bugs or vulnerabilities are used to penetrate networks and systems for financial benefits. This motivates cyber-criminals to exploit vulnerabilities to reach private and secret information, take down network servers, send spams and steal bank accounts. Intel Security estimates the annual cost to the global economy from cybercrime was more than $400 billion (McAfee Labs, 2015). To cope with the thousands of new malware samples that are discovered every day, security companies and analysts use different techniques.

In general, two common approaches to analyze malware samples are: static and dynamic. Static methods are the most popular approach to identify files. The approach statically traces executable files to obtain the sequence of bytes or instructions which are similar within a family of malware samples (Szor, 2005). Static analysis presents information about programs control, data flow, data dependency and other statistical specifications without actually executing the binary.

Static approaches can observe the entire execution path of the binary code and thus they characterize malware capabilities more accurately. Another advantage is lack of execution overhead. In contrast, they are unable to detect the malwares which utilize anti-antivirus detection techniques like: run-time packing, anti-reversing and anti-disassembly techniques (Yason, 2007). Polymorphic or metamorphic malware commonly use these techniques to evade detection. Evading techniques such as encryption, compression, garbage code insertion and code permutation cannot be easily detected with regular static methods (Hu, 2011).

Modern antivirus products use different static analyses heuristics such as: unpacks and statistical analyzers to counter obfuscations techniques. Definitely these heuristics are competitive with dynamic

* Corresponding author. Tel.: +98 7136133569; fax: +98 7136474605.
E-mail addresses: zsalehi@cse.shirazu.ac.ir (Z. Salehi), sami@shirazu.ac.ir (A. Sami), ghiasi@cse.shirazu.ac.ir (M. Ghiasi).

http://dx.doi.org/10.1016/j.engappai.2016.12.016
Received 3 April 2015; Received in revised form 30 November 2015; Accepted 15 December 2016

analysis for common and known malware families. However some of the heuristics that deal with obfuscations are NP hard (Moser et al., 2007). Thus, dynamic analysis also is used as a complement to improve malware detection (Cesare et al., 2013).

In dynamic method, a binary is run on a controlled or virtual environment and the run-time behavior of the binary is monitored. The behaviors are collected to analyze and detect malware samples. Behavioral methods are resource-intensive and are only able to detect behaviors which are observed at the run-time; so it may require stimulation inputs to explore all code segments (which could be the reason that some samples produced insufficient trace data). The task of program running in a controlled environment for a determined time is very resource/time consuming. Additionally, some of the malware detect environment and so do not execute or postpone their malicious intent (Bayer et al., 2009). The main advantage of dynamic approaches is the low-level mutation techniques such as run-time packing or obfuscation techniques that do not impress the behavioral features. Also dynamic approaches give the actual information about the control and data flow. Therefore researchers have used dynamic techniques to complete static analysis in malware detection.

Our mentality is based on the hypothesis that Application Programming Interface (API) alone may not represent intend of the operations that the function does. In other words, members of the same family of malware when they call the same APIs with similar arguments and return values they perform the same task. In previous work (Salehi et al., 2014), arguments and/or API name are used as feature to detect malicious from non-malicious applications. In this work, we are looking to introduce some features that could detect zero day attacks automatically and show extracted features on a dataset have discriminating capability. To justify the idea, we used a test set that contains new variant and new families of malware, not present in the training set. Feature sets are generated on combination of API names, their respective return values and/or input arguments in the run-time. The results show the proposed features of this work outperform the previous work and using the return value along with API call could be effective to identify new behaviors.

The rest of the paper is as follows: The related work and a background of other researches are discussed in Section 2. Section 3 presents the proposed malware detection system, payload executable (PE)'s behavioral monitoring, feature generation based on APIs and their dependencies and the learning phase. Datasets and the empirical evaluation of the method are given in Section 4. Finally, Section 5 concludes this paper.

2. Related works

Malware detection is a hot research topic only approaches presenting feature extraction techniques are introduced here. First some methods that deployed static features are described and then dynamic features are explained.

Walenstein et al. (2010) used a static method which extracted 4-grams features from PE header and body information to distinguish benign from malicious files. Shankarapani et al. (2011) extracted API sequences that appear frequently in a number of malware and applied similarity measure for the sequence. Tahan et al. (2012) removed the function libraries constructed by benign files from those which appear in malware as segment threats. They calculated segment entropy and extracted 3-grams Opcode for each segment. Baldangombo et al. (2013) selected subset of static features by calculating frequencies of Dynamic Link Libraries (DLLs), APIs and PE header feature. Information gain feature selection and classifiers techniques were applied to detect malicious files. Alazab et al. (2014) calculated Opcode frequency statistics of inspected PE files. They applied proposed hybrid wrapper-filter based feature selection.

Faruki et al. (2012) disassembled instructions of a binary program and captured the sequence of API calls to generate Control Flow Graph (CFG) statically. Macedo and Touili (2013) constructed malicious trees statically based on system functions or parameter values. Edge-label characterized the data flow between functions. Alam et al. (2014) divided an assembly program into smaller functions and generated a set of CFGs corresponding to separate functions of a program. Mehra et al. (2015) disassembled the dataset and generated CFGs that illustrate the flow of code segments. The selected feature set is used to create the feature vector and calculated Cosine similarity measure.

As mentioned, static methods cannot easily detect malware that uses evasion techniques; furthermore adding some fake API calls in the header of executable can make static malware detection ineffective. Therefore, dynamic analysis is needed as a complement for static techniques (Comparetti et al., 2010).

Plenty of dynamic works represent the behavior as a graph. Christodorescu et al. (2008) modeled system calls and their input/output arguments using the dependency graphs. They used the difference of the malware and benign sub-graphs to detect malware. Park et al. (2013) extracted behavioral graph of kernel objects and their dependencies for a group of instances in each family instead of finding a pattern for each sample. Zeng et al. (2013) collected program's control flow graph and information such as control structures, memory addresses and accesses, and safety information. Wüchner et al. (2014) translated communication between different systems entities such as processes, sockets, files or system registries as graphs to introduce a detection system based on quantitative data flow model. Generally, graph mining is complicated (Skaletsky et al., 2010; Macedo and Touili, 2013) and could not represent the samples which invoke a small set of system call.

Some dynamic approaches used API calls and their related properties to model behavior of samples. Ahmed et al. (2009) classified malwares by combining both the spatial (argument) and the temporal (API sequence) features. API calls that related to memory management actions and files I/O categories are monitored. This approach is computationally intensive. Tian et al. (2010) considered each API Call and other extracted features as string information. Ghiasi et al. (2013) assume that behavior of each binary can be represented by the values of register contents in its run-time. Ahmadi et al. (2013) used iterative system call pattern mining. They believed that repetitive actions on data sequences are often used by malware writers, especially some well-known loops performing decryption or encryption and infection.

Rieck et al. (2011) model sample behavior based on 2-gram features through system calls and their arguments by using prioritizing arguments. They identify novel classes of malware with similar behavior and assigning unknown malware to these discovered classes. Ravi and Manoharan (2012) used 4-grams to model API call sequences. By comparison of the average confidence of all 4-grams; samples are classified as malware or benign class. Cheng et al. (2013) used Windows API function call, its parameter and the corresponding value as a behavior feature sequence to classify malicious binaries by using information retrieval theory. Van Nhuong et al. (2014) focused on register and memory values as a semantic set. Then semantic set are used as a 3-gram input of Naïve Bayes classification. Piyanuntcharatsr et al. (2015) transformed reference dataset into the hexadecimal code to calculate the statistical features. Each statistical values were considered as 1, 2, 3 g features and counted feature frequency to model in Weka. Table 1 presents results from some various researches.

All mentioned related research, presented their works using API calls, their arguments and return-values in different ways. Some of them either used graph flows to present data/control between invoked API calls or used n-gram features or used string information to characterize the behaviors of binaries. Each method could have advantages and disadvantage. For example, some of methods that use graph matching to find the existing similarities between graphs, is time and space consuming because graph mining methods are NP-complete. Some works use small dataset or the extracted features are

Table 1
Comparison of several malware detection methods.

Study	Analysis Type	FP	Detection Rate	Feature	Representation
Mehra et al., 2015	Static	–	99.1%	flow of code segments through API call graphs	CFG
Baldangombo et al., 2013	Static	2.7%	99.6%	PE header information, DLL names and API names	String information
Tahan et al., 2012	Static	0.6%	98.6%	common segments (libraries and resources that originated from the development platform)	N-gram
Walenstein et al., 2010	Static	1.1%	99.2%	PE header and body information	N-gram
Wüchner et al., 2014	Dynamic	1.6%	96%	communication between different system entities (processes, sockets, files or system registries)	Data Flow Graph
Park et al., 2013	Dynamic	0	> 80%	kernel objects and their dependencies	Graph
Ahmadi et al., 2013	Dynamic	11.9	98.1(AUC)	executables' API call	API sequences and iterative patterns
Ghiasi et al., 2013	Dynamic	4.5%	96%	register values	Binary vector of Features
Ravi and Manoharan, 2012	Dynamic	–	90%	API call sequences	N-gram
Rieck et al., 2011	Dynamic	–	99.7%	system calls and their arguments	N-gram, Binary vector of Features
Anderson et al., 2011	Dynamic	–	96.41%	instruction traces	N-gram, CFG
Tian et al., 2010	Dynamic	2%	97.3%	API calls and their parameters	Binary vector of Features

fitted to their dataset. Thus these features are not extendable to use to others. In contrast to the problems of related works, MAAR method introduces a new method to generate features that is independent of dataset. MAAR generates a small set of robust features based on the combination of arguments and return values.

3. MAAR system

The overview of proposed system is shown in Fig. 1. This system has three components: in the first component, PE binaries are run and their behaviors are monitored under an in-house developed tool. API calls, input and output arguments and return values are monitored by the tool. Monitoring results are preprocessed and feature sets are generated based on combinations of API calls, and the corresponding arguments and/or return values of the APIs. Since a large set of features are generated, the set is reduced by feature selection techniques. Finally models are constructed to classify malware from benign samples. In the third phase, the discriminative ability of the constructed model will be tested using binaries that are not seen in the training phase. The mentioned phases are described as follows.

3.1. PE'S behavioral monitoring

In this section PE's behavioral monitoring is described. Our developed tool, environment configuration, the file execution process and log collection are explained in Section 3.1.1. The preprocessing procedure explains the reason of some binaries not to execute in controlled environment (Section 3.1.2). These files do not show their activities, so removed in the preprocessing. The importance of API calls and their dependent parameter are explained briefly in Section 3.1.3. In Section 3.1.4, feature generation phase, explains the combination of APIs and their arguments or return value and also defines our idea that is behind it.

3.1.1. Developed in-house tool

The "in-house" developed tool consists of a virtual machine, a hooking tool and a logging system. This tool analyzes the binary files and monitors their behaviors. In-house tool monitors API calls, input and output arguments, return values, register values and etc. (Ghiasi et al., 2013). The whole process is illustrated in Fig. 2 and a trace report is demonstrated in Fig. 3.

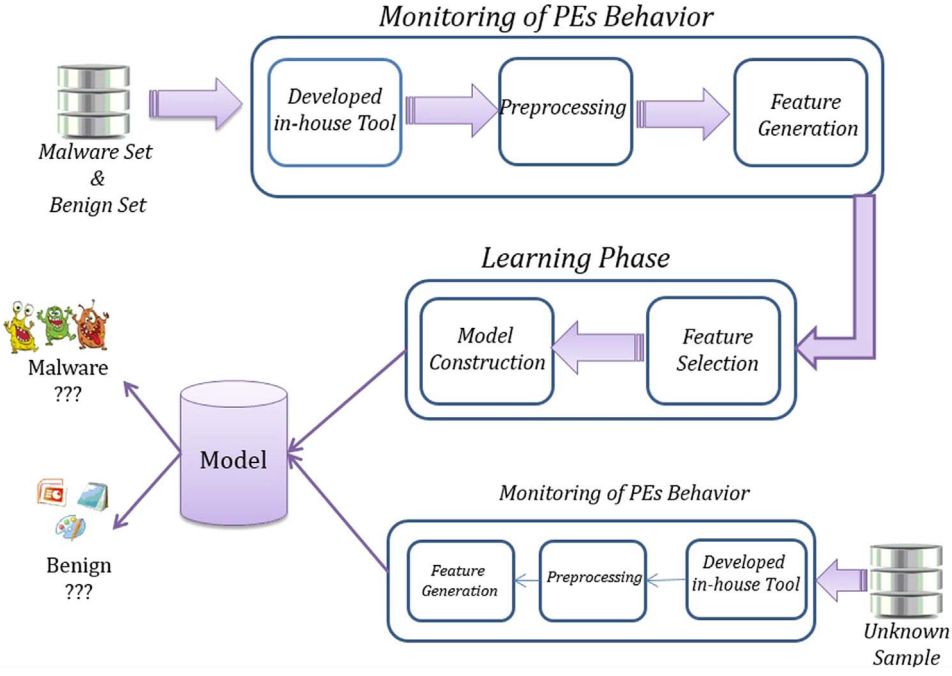

Fig. 1. MAAR system.

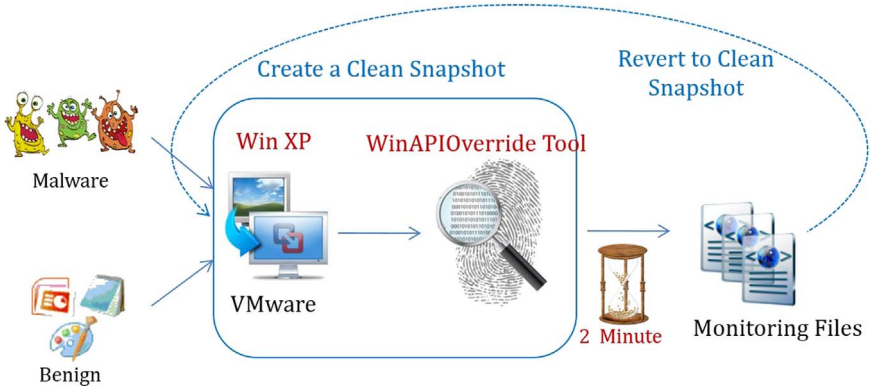

Fig. 2. Running the sample files under in-house tool.

Every monitoring technique such as hooking that is used in this research has its own advantages and disadvantages. For instance, CWSandbocx and Norman use inline hooking and kernel level methods respectively. Even though they are more advanced, they suffer from a series of disadvantages that can be evaded (Harbour, 2009; Chiu and Huang, 2010); a large number of malware could be created with polymorphic or metamorphic techniques that escapes detection by mentioned sandboxes. For instance, Norman does not monitor "GetSecurityDescriptorDacl, DeviceIoControl, WriteProcessMemory, CreateRemoteThread, NtQueryValueKey, RegOpenCurrentUser" API's.

Hooking technique has four main advantages that are used in this research. First, logs containing direct access to hardware can be monitored by monitoring DeviceIoControl API (DeviceIoControl function, 2015) which is not monitored in other techniques. Secondly, the selected hooking method monitors returned-values of the function calls which is a main topic of this paper. Thirdly, GetSecurityDescriptorDacl API, which is used in Duqu, Flame, Stuxnet (Chien et al., 2012) can be monitored by our tools which is not monitored in Norman. Finally, we have greatly benefited from WinAPIOverride32 that is freely available (Potier, 2015).

3.1.2. Preprocessing phase

After monitoring of PEs behaviors, some log files are removed from the analysis. These files which their logs are less than 70 KB, are not successfully executed so they are not considered in this phase either.

Several reasons exist that a file is not executed properly. The main reason is that some files have unrecognizable formats. Secondly, some files are dependent on other files or system services which are not provided. Also, it is possible that some files evade the dynamic techniques, if they detected the configured environment and self-terminated before executing their payload. Sophisticated malwares use several techniques to detect whether file is executed under a controlled environment. These techniques are: checking the files, registry keys, or processes that are specific to individual analysis tools; or examining existing variations in the semantics of CPU instructions or timing properties (Balzarotti et al., 2010).

3.1.3. Importance of system calls

Our model construction mentality is based on the hypothesis that API names alone may not represent intent of the operations that the function does. Same family members of malware when they perform the same task call same APIs with similar arguments. Basically, the main hypothesis of this research is the arguments and the return values of the same family members are the same for the same API and are different for different families. For exploring the mentality, several examples are given below.

Malware binaries invoke many system calls to execute their malicious payloads; they might try to create registry keys in the windows registry, modify certain parts of the system registry or rewrite the specific files and folders. Some malware files can propagate themselves or seek application files to reproduce themselves by infecting and merging into files of the application by calling specific APIs like: FindFirstFileA, CopyFileA, GetFileType and SetFilePointer (Belaoued and Mazouzi, 2014). For example the data flow between GetModuleFileName and CopyFileA API could establish the malicious behavior in the self-replication operation: The GetModuleFileName function is called with NULL as first parameter thus it will return the malware file path. The file path is used as input in the first parameter of a subsequent call to CopyFile. Also malware can copy itself to the Windows system directory rather than creating its own program directory in Program Files.

Some malware can gain non authorized user data level by cross exchanging data with running applications. Malware can also gain user privileges, auto run the malware when the system is starting up or insert an

Call	Ret Value
CreateFileW(lpFileName:0x0012EE00:"C:\WINDOWS\system32\wdmaud.drv",dwDesiredAccess: 0x0,dwShareMode: 0x1,lpSecurityAttributes:0x000000...	0x0000039C
CloseHandle(hObject: 0x0000039C)	0x00000001
LoadLibraryW(lpFileName:0x0012F038:"wdmaud.drv")	0x72D20000
LoadLibraryW(lpFileName:0x72D21250:"setupapi.dll")	0x77920000
CreateFileW(lpFileName:0x00181780:"\\?\root#system#0000#{3e227e76-690d-1...,dwDesiredAccess: 0xC0000000,dwShareMode: 0x0,lpSecurityAttri...	0x000003AC
DeviceIoControl(hDevice: 0x000003AC,dwIoControlCode: 0x1D8000,lpInBuffer: 0x00174F10: {00 00 00 00 00 00 00 00 05 00 00 00...,nInBufferSize: 0...	0x00000001
CloseHandle(hObject: 0x000003A4)	0x00000001
GetModuleFileNameA(hModule: 0x72D20000,lpFilename:0x0012F3E8:"C:\WINDOWS\system32\wdmaud.drv",nSize: 0x104)	0x0000001E
DeviceIoControl(hDevice: 0x000003AC,dwIoControlCode: 0x1D8004,lpInBuffer: 0x0017FC40: {00 00 00 00 00 00 00 00 01 00 00 00...,nInBufferSize: ...	0x00000001
CloseHandle(hObject: 0x000003A4)	0x00000001
DeviceIoControl(hDevice: 0x000003AC,dwIoControlCode: 0x1D8010,lpInBuffer: 0x0017FC40: {00 00 00 00 00 00 00 00 01 00 00 00...,nInBufferSize: ...	0x00000001
CloseHandle(hObject: 0x000003A4)	0x00000001
RegCloseKey(hKey: 0x00000398)	0x00000000
RegCloseKey(hKey: 0x00000394)	0x00000000
RegOpenKeyExW(hKey: 0x00000390,lpSubKey:0x76B481EC:"Drivers\wave",ulOptions: 0x0,samDesired: 0x9,phkResult: 0x0012F550: 0x00000394)	0x00000000

Fig. 3. A trace report of executable file.

entry in registries using related APIs such as: RegOpenKeyExA, RegQueryValueExA, RegSetValueExA, RegCreateKey and RegSetValue (Belaoued and Mazouzi, 2014). In other example, files can be protected from being read or scanned by a variety of means. A program can simply call CreateFile on itself with the share permissions set to NULL. This prevents other applications from opening the file until it is closed by the program (Chien, 2005).

Therefore the trace of API functions and their parameters is a remarkable source for modeling malicious behavior. For example, RegCreateKey and RegReplaceKey may appear in several variations. The use of a registry is very common in windows applications and does not mark a malicious intent by itself, but in combination with certain parameter values an invoking process can produce a malicious behavior. If an executable attempts to replace the content of a registry entry that determines the programs executed during the boot process or is equipped by a parameter that contains a known malicious program name or URL, it can be classified as malicious (Zamir et al., 2004).

3.1.4. Feature generation phase

API calls, input arguments and return values are used to construct a feature set modeling malicious and benign behaviors. Features of these datasets are considered to be binary valued features. The first feature set is a representation of called APIs along with their return values called "API-RET" from this point on in this paper. The second feature set is called "API-ARG" that represents API name alongside with their arguments. Finally, the third dataset is a combination of API-RET and API-ARG therefore we named it "API-ARG-RET".

Let us consider the CopyFileA (2015), GetModuleFileName (2015) API call. Table 2 expresses the CopyFileA and GetModuleFileName parameters.

BOOL WINAPI CopyFileA (lpExistingFileName, lpNewFileName, bFailIfExists).

DWORD WINAPI GetModuleFileName (hModule, lpFilename, nSize).

In CopyFileA, if *"bFailIfExists"* is ***TRUE*** and the new file specified by *"lpNewFileName"* already exists, the function will fail. In contrast if the argument is ***FALSE*** and the new file already exists, the function will overwrite the existing file and will succeed. Malware that exploits this API, overwrites pre-defined files.

The below example explains a scenario of a malicious behavior to establish the self-replication process. A typical instance of such behavior is a program that copies its own binary representation into another file. The attacker program discovers and stores its file path into a memory address by calling the GetModuleFileName function with ***NULL*** as first parameter. The second parameter of the GetModuleFileName is defined as an output (If the first parameter is ***NULL***, GetModuleFileName retrieves the path of the executable file of the current process). The parameter is outputted by GetModuleFileName call and is used as input of CopyFile to infect another file as first parameter.

GetModuleFileNameA(hModule: NULL, lpFilename: "Trojan.Win32.CrazyCD.exe", nSize: "0×18"): Return-Val: Trojan.Win32.CrazyCD.exe "

CopyFileA (lpExistingFileName: "Trojan.Win32.CrazyCD.exe", lpNewFileName: "C:/Windows/system32/dwwin.exe", bFailIfExists: 0×0): Return-Val: 0×1.

When a malicious binary calls the above APIs, "*dwwin.exe*" which is placed in "*C:/Windows/system32/*" with its file will overwrite "Pacman.exe". Let us to consider CopyFileA to extract mentioned feature sets as follows; the feature "*CopyFileA.RetVal.0×1*" is constructed as an API-RET feature where API name is the first element, "*RetVal*" is the indicator of returned value and finally " *0×1*" is its returned value. API-ARG features include the following features for the three parameters of CopyFileA API:

1. CopyFileA.1.Trojan.Win32.CrazyCD.exe
2. CopyFileA.2.C:/WINDOWS/system32/dwwin.exe
3. CopyFileA.3.0×0

In API-ARG feature set, API name is the first element, the order of input argument is the second element and the corresponding argument value is the third element of the feature. Third feature set (API-ARG-RET) is regarded as the combination of first and second feature sets. In this example, this feature set has three members. Each member consists of 5 elements.

1. CopyFileA.1.Trojan.Win32.CrazyCD.exe.RetVal.0×1
2. CopyFileA.2.C:/WINDOWS/system32/dwwin.exe.RetVal.0×1
3. CopyFileA.3.0×0.RetVal.0×1

By proposing these three feature sets we try to cover all possible malicious activities in the modeling phase; considering *CopyFileA*, API name alone does not seem to be discriminative in the term of recognizing malicious activity. Regarding the API-RET, knowing that whether a copy operation is successful or not, may be part of factors to distinguish malicious from benign activities. Our hypothesis is that the API-ARG feature set provides more accurate models by having the knowledge about the source and destination of a copy operation. Regarding the third feature set, using API name seems to be important while knowing the location of files that has been copied and the return value. For example, if a file is copied in a path other than *"C:/Windows"*, it is possible that this API did not harm the system, but overwriting a system file (for example like those that are located in *"C:/Windows"*) can be important. The feature "*CopyFileA.2.C:/ Windows/system32/dwwin.exe.RetVal.0×1*" indicates that overwriting operation is done successfully according to the return value *'0×1'*. The issue also reveals the importance of third feature set API-RET-ARG. It seems introduced three feature sets in this paper could provide higher

Table 2
Description of CopyFileA and GetModuleFileName parameter.

CopyFileA API		
Description	***Parameter***	
Name of an existing file	*LpExistingFileName*	**First Parameter**
Name of the new file	*LpNewFileName*	**Second Parameter**
Corresponding arguments have Boolean range which means they can be set to true or false (1 or 0)	*bFailIfExists*	**Third Parameter**
If the function succeeds, the return value is nonzero. If the function fails, the return value is zero		**Return Value**
GetModuleFileName API		
A handle to the loaded module, if *NULL*, *GetModuleFileName* retrieves the path of the executable file of the current process	*hModule*	**First Parameter**
A pointer to a buffer that receives the fully qualified path of the module	*lpFilename*	**Second Parameter**
size of the *lpFilename* buffer	*nSize*	**Third Parameter**
If the function succeeds, the return value is the length of the string that is copied to the buffer. If the function fails, the return value is zero		**Return Value**

accuracy than earlier datasets that only consist of API name.

It may seem that the generated dataset API-RET-ARG is a super set of the first two generated feature sets. However, the former only contains a certain combinations of the latter two. Thus it is not a superset of the API-RET and API-ARG. Continuing with the above example, CopyFileA can have a certain return value when the destination is "C:/WINDOWS". Even if the API returns both 0 and 1, conceptually the meaning of these values are different based on the destination indicated as a parameter.

For all system calls, the values of input parameters and the return value can be extracted and three categories of features are constructed. After constructing features, presence of each feature in every binary file is checked and shown in a vector. If the selected feature is available in the log file, the value of this binary feature is set to "1" otherwise it is set to "0".

3.2. Learning phase

Any classification process contains two phases: feature selection and model construction. In feature selection the search space is reduced while preserving the discriminative and useful features. In the model construction, classifiers are built.

A lot of parameters are monitored in behavior monitoring. Preprocessing reduces generalization error of the learned models because irrelevant and redundant features are removed. Too many features are generated in the monitoring stage. For instance a polymorphism technique called junk code insertion creates a lot of unnecessary features that are not discriminative for malware detection. Moreover, several misleading features may be inserted into malicious binaries to make their detection difficult.

To show the effectiveness of our generated features, no major attempt is used to find the optimal or best set of features. By investing more attention on the feature selection techniques definitely better results will be obtained. In the first stage, Fisher score is used just for its speed. Fisher score (Duda et al., 2000) selects the most discriminative features. This score is defined as:

$$Fr=\frac{\sum_{i=1}^{c} n_i(\mu_i-\mu)^2}{\sum_{i=1}^{c} n_i \sigma_i^2} \quad (1)$$

Where n_i is the number of data samples in class i, μ_i is the average feature value in class i, the standard deviation of the feature values in class i is shown by σ_i, and is the average feature value in the whole dataset.

In the next stage, the well-known feature selection algorithm called Support Vector Machine based on Recursive Feature Elimination (SVM-RFE) with 10-fold cross validation is utilized to select a smaller set of features (Guyon et al., 2002). Selected features are considered to be much more discriminative, and lead to the largest margin of class separation based on SVM-RFE. This provides an efficient method which decreases the processing time of classification algorithms and removes the irrelevant features.

If the number of features becomes more than samples the classification algorithm may trap into a phenomenon called over-fitting. To avoid over-fitting, we try to reduce features. So two rounds of feature selection along with 10-fold cross validation are utilized. When the features are selected, presence of each feature in every binary file will be checked.

Classification algorithms help to discover discriminative patterns in the datasets. Discovered patterns are used to classify unseen samples. The generated binary vectors from the previous stage are used as input to the classification algorithms. Several well-known classifiers such as Random forest (RF), J48 Decision Trees, Sequential minimal optimization (SMO), Bayesian logistic regression (BLR) implemented in Weka 3.6.6 are used in this study. Classifiers are evaluated using 10-fold cross validation procedure for all experiments to avoid over-fitting. The F-measure of each run is calculated. Finally the average of this measure is reported as evaluation criteria.

4. Evaluation phase

Datasets and distribution of malicious and benign samples that are used in this study, the process of API and DLLs selection and the monitored APIs and DLLs that are used to model the PEs behavior are explained in Section 4.1. The experimental evaluations of the proposed method and discussion about them are given in Section 4.2.

4.1. Data acquisition and essential API calls for detecting malicious behavior

Three datasets named *"First"*, *"Second"* and *"Total"* are used in this paper. Each dataset has malicious and benign samples and is collected from (Sami et al., 2010). Malicious samples include seven categories of Constructor, Trojan, Virus, Backdoor, and etc. The benign ones are windows system files and a wide range of portable benign tools. First dataset consists of 385 benign and 826 malicious samples. Second dataset is used to validate that generated features have discriminative power and are extendible regardless of the dataset. A second dataset consist of 974 benign and 2183 malicious samples other than First dataset. *"Total"* dataset is combination of First and Second dataset. *"Second"* dataset has "worm" samples, which is not included in First dataset and has more variants of the families compare to the First. As a result, Total dataset contains of 3009 malicious and 1359 benign files. For an easier comparison, the distributions of First and Total dataset are presented in Fig. 4. All experiments in this paper are performed on First, Second and Total datasets. The detailed list of examined executables and datasets can be found in the following URL: http://home.shirazu.ac.ir/~sami/malware/.

In this study, all API calls are not monitored; only a subset of 126 API calls from six most important DLLs is logged. The important DLLs are: advapi32.dll, kernel32.dll, ntdll.dl1, user32.dll, wininet.dll and

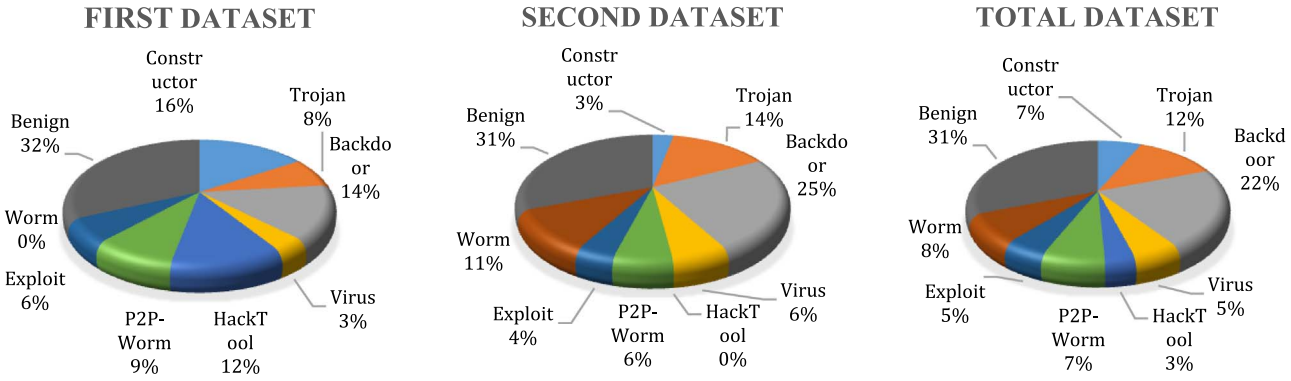

Fig. 4. Distribution of sample types in first, second and total dataset.

ws2_32.dll. These API calls are selected from our previous research, done by (Karbalaie et al., 2012) that are gathered based on the File System Access, System Information, Networking, Registry Access and Processes categories (Wagener, 2006).

4.2. Experimental and discussion

In this section to investigate the robustness and efficiency of generated feature, three set of experiments are performed. In the first experiment, malware detection accuracy, F-Measure, Recall, Precision, False Positive (FP) and Root Mean Square Error (RMSE) are measured based on First and Total dataset. Then these results are compared with several antiviruses. Since threats are increasing at an enormous rate, generating the features that are useful to detect different new set of malware is important. To inspect this issue in the second experiment, Section 4.2.2, the features of the "FIRST" dataset are used to create models on "Second" dataset. The detection results of the mentioned features are investigated to show whether features are dataset independent and discriminative enough on the "Second" dataset to classify malicious activity from benign on unseen data.

Comparison of different Malware detection algorithms is difficult. The researches presented in the related work section used different datasets (majority not pubic) and used different sandboxes (some did not even mention how the sandbox); others have recorded behavior of files in different formats. Thus, comparison would have become biased even if we would have implemented their algorithms. Stated differently, the results that we would have obtained by implementing other researchers work is not justifiable on our dataset and may raise several other misunderstanding. (Tahan et al., 2012) emphasizes the difficulties in making any comparison in Malware research. To present a verifiable comparison, we compared our method with some popular anti-viruses on the same dataset in first experiment. In addition, two works that used API alone on the same dataset are implemented to evaluate the MAAR feature sets. In third experiment, Tian et al. (2010) and (Ravi and Manoharan, 2012) methods are implemented. API and API sequence (API-SEQ) features are generated respectively based on these researches. These features detection capability practically compared to MAAR feature sets in Section 4.2.3 to investigate that which one is more discriminative in malware detection. The experiments results and discussion are presented in the following.

4.2.1. Malware detection accuracy

The most interesting question in the field of malware detection is the detection ability and accuracy of a newly generated method. So some experiments done to determine the detection capability of the generated feature sets. Also comparison between the results of the presented system and the results of three noted industrial antiviruses (McAfee, Avira and Kaspersky) are presented. Because nearly 30% of the dataset consists of benign files, the dataset is imbalanced; it is shown that precision, recall and F-Measure are proper measures for unbalanced datasets (Weng and Josiah, 2008). Accuracy for unbalanced dataset may be very misleading. For instance, in a recently published paper (Langerud and Lillesand, 2008) if all the samples were naively classified as malware irrespective of type, accuracy of above 90% would have been obtained.

Table 4
RMSE and False Positive Rate (feature form First D.S/ Test on First D.s).

	API-RET		*API-ARG*		*API-ARG-RET*	
	RMSE	***False Positive***	***RMSE***	***False Positive***	***RMSE***	***False Positive***
RF	0.188	0.031	0.149	0.034	0.153	0.026
J48	0.222	0.101	0.246	0.140	0.255	0.148
SMO	0.141	0.023	0.111	0.008	0.111	0.010
BLR	0.132	0.018	**0.086**	**0.003**	**0.091**	**0.008**

According to the method presented in Section 3, feature sets are extracted from First dataset and model is constructed on using 10-fold cross validation. The high precision, recall, F-measure and AUC values in Table 3 and low false positive rate in Table 4 indicate that all feature sets can detect malwares with high accuracy. Since "arguments" or "return value with arguments "can represent the functionality of an API call better than API_ARG feature sets, it is rational to have the highest F-measures and lowest false positive with API-ARG and API-ARG-RET feature set. Also the high values of AUC in Table 3 show that the models based on generated features have acceptable detection rate. The RMSE values of the conducted experiments are given in Table 4. The small RSME values show relativel low level of uncertainty.

Then we expanded our dataset; so all experiments are done on the Total dataset (using 10-fold cross validation). The following experiments, list new generated feature sets that are updated. The detection rates are illustrated in Table 5 and the false positives are displayed in Table 6.

In the experiment that conducted on First dataset (Table 3 and Table 4), API-ARG and API-ARG-RET feature sets have high detection rate and obtain approximately same F-Measure. Also the low values of FP and RMSE show that the error is low in both of them. In the subsequent experiment that test on Total dataset, the F-Measure of API-RET and API-ARG-RET feature sets is almost similar and has the same error rate. Based on the two performed experiment where test and train are from First and Total, API-ARG-RET performs very well in either case.

Since the sandbox, tools and dataset used in other researches are not publically available. It is assumed that commercial antivirus applications have higher detection capability than previously published researches, the comparison of our system with commercial AV-applications is presented. Table 7 shows the comparisons of the MAAR method with different antivirus application on First and Total datasets. The system based on API-ARG-RET feature set achieves higher F-Measure compare to other antivirus applications.

4.2.2. Discriminative ability, irrespective of data

In this experiment investigated whether a few extracted features on a dataset have discriminating capability, irrespective of the dataset. The feature sets are not dataset dependent. Based on the features extracted from a small dataset, the model is constructed on a much larger dataset with more malware families. The use of cross validation in feature selection and classification guarantees that, the test data is not used in training. In addition the features extracted from the First dataset are

Table 3
The Precision, Recall, F-Measure and AUC results on 'First' dataset when three feature sets are generated based on 'First' dataset.

	API-RET				*API-ARG*				*API-ARG-RET*			
	Precision	***Recall***	***F-M***	***AUC***	***Precision***	***Recall***	***F-M***	***AUC***	***Precision***	***Recall***	***F-M***	***AUC***
RF	0.985	0.954	0.969	0.990	0.984	0.977	0.981	0.995	0.988	0.973	0.980	0.995
J48	0.954	0.969	0.961	0.940	0.937	0.969	0.952	0.935	0.933	0.969	0.951	0.924
SMO	0.989	0.982	0.985	0.979	0.996	0.985	0.991	0.989	0.995	0.987	0.991	0.988
BLR	0.991	0.983	0.987	0.982	**0.999**	**0.990**	**0.995**	**0.994**	**0.996**	**0.992**	**0.994**	**0.992**

Table 5
The Precision, Recall, F-Measure and AUC results on 'Total' dataset when three feature sets are generated based on 'Total' dataset.

	API-RET				*API-ARG*				*API-ARG-RET*			
	Precision	***Recall***	***F-M***	***AUC***	***Precision***	***Recall***	***F-M***	***AUC***	***Precision***	***Recall***	***F-M***	***AUC***
RF	0.968	0.969	0.969	0.984	0.960	0.961	0.961	0.981	0.968	0.975	0.971	0.989
J48	0.958	0.969	0.963	0.954	0.959	0.952	0.955	0.96	0.965	0.969	0.967	0.964
SMO	0.978	0.977	0.978	0.964	0.973	0.969	0.971	0.954	0.981	0.983	0.982	0.970
BLR	**0.979**	**0.98**	**0.98**	**0.967**	0.974	0.970	0.972	0.957	**0.979**	**0.984**	**0.981**	**0.968**

Table 6
RMSE and False Positive Rate (Feature from Total D.S / Test on Total D.S).

	API-RET		*API-ARG*		*API-ARG-RET*	
	RMSE	***False Positive***	***RMSE***	***False Positive***	***RMSE***	***False Positive***
RF	0.194	0.071	0.205	0.088	0.184	0.072
J48	0.215	0.093	0.232	0.091	0.205	0.079
SMO	0.176	0.049	0.200	0.060	0.159	0.043
BLR	**0.167**	**0.046**	0.195	0.057	**0.162**	**0.048**

Table 7
F-MEASURE of system in comparison to some of the updated antiviruses.

	Antivirus Software			*MAAR Method*		
	McAfee	***Kaspersky***	***Avira***	***API-RET***	***API-ARG***	***API-ARG-RET***
First Dataset	**0.944**	0.896	0.920	0.987	0.995	**0.994**
Total Dataset	0.959	0.905	**0.969**	0.980	0.972	**0.981**

used to construct models on the Second dataset which is 2.6 times larger than the former one with completely differnt samples.

In comparison with the First dataset, Second dataset has more benign and malicious samples; malwares from new categories and families with new behaviors exist in the Second dataset. In other hand, some families that do not exist in First dataset are presented in Second dataset and so new behaviors are observed. As an illustration, worm category only exists in the Second dataset but not in the First dataset that features are extracted from. This is a vital indication to investigate the effectiveness of generated features on unseen families and samples. Second dataset has different samples with different behaviors to check whether experiments can show discriminatory ability of extracted features.

Efficacy of generated features from the First dataset is evaluated as an indication of robustness on Second dataset. The detection rate of models built on features extracted from First dataset and tested on Second is illustrated in Table 8. As observed, these features are robust and identified unseen samples in Second dataset. The results of Weka classifiers using 10-fold cross validation showed that in the best case, Bayesian logistic regression obtains an accuracy of 96% on API-RET feature set. While the accuracy of the API-ARG-RET feature set is slightly lower (about 3.3%). The RMSE and false positive rate of these is shown in Table 9. As you can see in this experiment, API-RET false positive rate increased about 7% while the RMSE value increased about 10% in comparison with two other feature sets. These results demonstrate the discriminative power of API-RET feature set.

Table 9
RMSE and False Positive Rate (Feature from First D.S / Test on Second D.S).

	API-RET		*API-ARG*		*API-ARG-RET*	
	RMSE	***False Positive***	***RMSE***	***False Positive***	***RMSE***	***False Positive***
RF	0.229	0.121	0.285	0.192	0.269	0.141
J48	0.230	0.098	0.302	0.218	0.301	0.176
SMO	**0.226**	**0.073**	0.321	0.156	0.319	0.158
BLR	0.236	0.089	0.326	0.159	0.317	0.155

Empirical results suggests API-RET feature set performs better when the experiment is extended to a larger datasets, whereas API-ARG-RET performs better when the model is developed and tested on the same dataset but we used 10-fold cross validation in all experiments. The phenomena can be explained as follows: the behaviors could be modeled better based on the combination of the arguments and return values, but note that input arguments may have many values and a large span may exist between new families with different behaviors, whereas return values have a narrower span and are more stable in extending the dataset. In other words, API-RET feature set gives a more general model of the behavior in contrast to API-ARG-RET feature set and models the behavior more specifically. Thus, in case of incremental learning, API-ARG-RET may provide more robust results than other features that are built on fewer parameters. Whereas it would be more reasonable to use API-RET when the feature sets are not updated. These features could detect some samples of the new families automatically. Antivirus software usually need a period of time to detect and analysis the new samples signatures. Along with the great detection capacity, MAAR method could also avoid the systems to be contaminated by the new malware types in their zero day attacks.

4.2.3. The best generated feature set among API, API-SEQ, API-RET, API-ARG and API-ARG-RET

As mentioned before, since sandbox, tools, dataset and behavioral

Table 8
The Precision, Recall, F-Measure and AUC results on 'Second' dataset when three feature sets are generated based on 'First' dataset.

	API-RET				*API-ARG*				*API-ARG-RET*			
	Precision	***Recall***	***F-M***	***AUC***	***Precision***	***Recall***	***F-M***	***AUC***	***Precision***	***Recall***	***F-M***	***AUC***
RF	0.946	0.950	0.948	0.973	0.916	0.935	0.925	0.938	0.937	0.928	0.932	0.951
J48	0.956	0.953	0.955	0.961	0.906	0.939	0.922	0.913	0.922	0.923	0.925	0.909
SMO	**0.967**	**0.958**	**0.963**	**0.943**	0.930	0.921	0.925	0.882	0.929	0.923	0.926	0.882
BLR	0.960	0.959	0.96	0.935	0.928	0.918	0.923	0.879	0.930	0.923	0.927	0.884

Table 10
Comparison of three feature set with only API name (BLR classification) and API-SEQ method.

	First Dataset					*Second Dataset*				
	API	***API-SEQ***	***API-RET***	***API-ARG***	***API-ARG-RET***	***API***	***API-SEQ***	***API-RET***	***API-ARG***	***API-ARG-RET***
Precision	**0.920**	0.100	0.991	**0.999**	0.996	0.872	**0.998**	**0.960**	0.928	0.930
Recall	**0.937**	0.892	0.983	**0.990**	0.992	0.762	**0.591**	**0.959**	0.918	0.923
F-M	**0.929**	0.943	0.987	**0.995**	0.994	0.813	**0.742**	**0.960**	0.923	0.927
F-P	**0.174**	0.177	0.018	**0.003**	0.008	0.251	**0.408**	**0.089**	0.159	0.155

outputs of the related research experiments are not publically available, it is not possible to compare the results of them and MAAR method (Tahan et al., 2012). Although the extracted API features from in-house developed tool are different from other tools, we tried to implement two methods that use only API call (Tian et al., 2010) and API call sequence (API-SEQ) based on features (Ravi and Manoharan, 2012). The method of (Tian et al., 2010) and (Ravi and Manoharan, 2012) are explained briefly in related works previously. For this comparison the First dataset are used as training phase and the First and Second dataset are used to test malware detection system using the method of these two papers. Since the best classifier results for our dataset belongs to Bayesian logistic regression using 10 fold cross validation, the results of BLR are compared with two implemented methods that mentioned previously. The comparison results are shown in Table 10.

In the Table 10, first group shows the results of the First dataset and the second group is dedicated to the Second dataset results that their features extracted from First one. Column labeled "*API*" stands for a feature set that only consists of API name alone as the feature set. Second column labeled "*API-SEQ*" shows result of method that use API sequence feature set (Ravi and Manoharan, 2012). Third, fourth and the last column in each group are dedicated respectively to our newly proposed features.

Table 10 shows the effectiveness of MAAR method that introduces API-RET feature set. An appropriate malware recognition approach is not only expected to be capable of distinguishing almost all unwanted threats but also envisage with a reasonable false alarm ratio. Regarding depicted results in the Table 10 the worst feature set is the API-SEQ and API name in terms of false alarm rate. Also the results show when the features are not updated API-RET has a better detection capacity compared to other feature sets. In comparison with the API feature set MAAR method shows a significant and consistent improvement in F-measure. The accuracy improves about 15% while the false alarm rate decreases about 17%.

5. Conclusion

In this paper a method called MAAR to produce robust and scalable feature sets to perform dynamic malware behavior analysis was presented. Features were generated based on the name of the API calls along with each of the argument and/or the return value of the API call at the execution time. Then, two phases of feature selection were utilized to reduce the number of features, in order to speed up the time of processing and increasing the detection rate. The behaviors of each binary were modeled based on the generated feature. To validate discriminative ability of proposed feature irrespective of dataset, two datasets one 2.6 times larger than the other and had new type of malware and benign with different behaviors had been generated. Results demonstrated that, the created features were highly discriminative and robust regardless of the dataset which are used. In future, models that will be able to add new features to itself and easily update itself will be developed. Those models could detect new variant of malware without significant reduction in accuracy.

Acknowledgments

Authors would like to thank Mr. Abdullah Khalili, Javad Kamiabi, Alireza Hoseini, Mansour Ahmadi and Ms. Shabnam Salehi for their help during all stages of this research. This research was conducted in the Department of Electrical Engineering and Computer Science of Shiraz University and the authors would like to express their sincere thanks to **Iran Telecommunication Research Center** (ITRC) for their supports.

References

Ahmadi, M., Sami, A., Rahimi, H.,, Yadegari, B., 2013. Malware detection by behavioural sequential patterns. Comput. Fraud and Secur. 8, 11–19.
Ahmed, F., Hameed, H., Shafiq, M.Z., Farooq, M., 2009. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In: AISec '09: Proceedings of the 2nd ACM Workshop on Secur. and Artificial intell. ACM, Chicago, Illinois, USA, pp. 55–62.
Alam, S., Horspool, R.N., Traore, I., 2014. MARD: a framework for metamorphic malware analysis and real-time detection. In: 2014 IEEE 28th International Conference on Advanced Inf. Networking and Appl., pp. 480–489.
Alazab, M., Huda, S., Abawajy, J., Islam, R., Yearwood, J., Venkatraman, S.,, Broadhurst, R., 2014. 2014. A hybrid wrapper-filter approach for malware detection. J. Netw. 9 (11), 2878–2891.
Anderson, B., Quist, D., Neil, J., Storlie, C.,, Lane, T., 2011. Graph-based malware detection using dynamic analysis. J. Comput. Virol. 7 (4), 247–258.
Baldangombo, U., Jambaljav, N., Horng, S.J., 2013. A static malware detection system using data mining methods. arXiv Prepr. arXiv 1308.2831.
Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., Vigna, G., 2010. Efficient detection of split personalities in malware. In NDSS 2010: Proceedings of the 17th Annu. Symp. Netw. *Distrib.* Syst. Secur. San Diego, CA, USA.
Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E., 2009. Scalable behavior-based malware clustering. In: NDSS 2009: Proceedings of the 16th Annu. Symp. Netw. Distrib. Syst. Secur. San Diego, CA, USA, pp. 8–11.
Belaoued, M.,, Mazouzi, S., 2014. 2014. Statistical study of imported APIs by PE type malware. In advanced Netw. *Distrib.* Syst.s Appl. (INDS). International Conference on IEEE, 82–86.
Cesare, S., Xiang, Y.,, Zhou, W.L., 2013. Malwise–An effective and efficient classification system for packed and polymorphic malware. IEEE Trans. Comput. 62 (6), 1193–1206.
Cheng, J.Y.C., Tsai, T.S., Yang, C.S., An information retrieval approach for malware classification based on windows API calls International Conference on Mach. Learn. Cyber. (ICMLC 2013), vol. 4, pp. 1678–1683.
Chien, E., 2005. Techniques of adware and spyware. In: Proceedings of the 15th Virus Bulletin Conference Dublin Ireland.Vol. 47.
Chiu, J.Huang, W., 2010. 0Box Analyzer: after dark runtime forensics for automated malware analysis and clustering. In: Proceedings of the 18th Annu. DEFCON Conference Las Vegas, USA.
Christodorescu, M., Jha, S., Kruegel, C., 2008. Mining specifications of malicious behavior, In: Proceedings of the 1st India Softw. Eng. Conference ACM 2008, Dubrovnik, Croatia, pp. 5–14.
Comparetti, P.M., Salvaneschit, G., Kirdai, E., Kolbitsch, C., Kruegel, C.,, Zanero, S., 2010. Identifying dormant functionality in malware programs. Proc. IEEE Symp. Secur. Priv., 61–76.
CopyFile function, 2016. Microsoft Msdn, [On-line].Available electronically at⟨http://msdn.microsoft.com/en-us/library/windows/desktop/aa363851%28v=vs.85%29.aspx⟩
Chien, E., OMurchu, L., Falliere, N., 2012. W32. Duqu: The precursor to the next Stuxnet. Presented as Part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET). Version1.4. 2015.
DeviceIoControl function, 2016. Microsoft Msdn, [On-line].Available electronically at⟨http://msdn.microsoft.com/en-us/library/windows/desktop/aa363216%28v=vs.85%29.aspx⟩
Duda, R.O., Hart, P.E., Stork, D.G., 2012. Pattern Classification 2nd ed. John Wiley & Sons, New York.
Faruki, P., Laxmi, V., Gaur, M.S., Vinod, P., 2012. Mining control flow graph as API call-grams to detect portable executable malware. In: Proceedings of the 5th International Conference on Secur. Inf. Netw. SIN 12, ACM, pp. 130–137.

GetModuleFileName function, 2016. Microsoft Msdn, [On-line]. Available electronically at ⟨https://msdn.microsoft.com/en-us/library/windows/desktop/ms683197%28v=vs.85%29.aspx⟩

Ghiasi, M., Sami, A.,, Salehi, Z., 2013. DyVSoR: dynamic malware detection based on extracting patterns from value sets of registers. ISC Int. J. Inform. Secur. 5 (1), 71–82.

Guyon, I., Weston, J., Barnhill, S.,, Vapnik, V., 2002. Gene selection for cancer classification using support vector machines. Mach. Learn. 46 (1–3), 389–422.

Harbour, N., 2009. Win at Reversing - API tracing and sandboxing through inline hooking. In 17th Annu. DEFCON Conference BlackHat, USA.

Hu, X., 2011. Large-scale malware analysis, detection, and signature generation Ph.D. Dissertation. Univ. of Michigan, , Michigan, United States.

Karbalaie, F., Sami, A.,, Ahmadi, M., 2012. Semantic malware detection by deploying graph mining. Int. J. Comput. Sci. Issues (IJCSI) 9 (1), 373–379.

Langerud, T.,, Lillesand, J.V.., 2008. PowerScan: A Framework For Dynamic Analysis And Anti-virus Based Identification Of Malware M.S. thesis). Norwegian Univ. of Sci. and Technol. Dept. of Telematics, Norway.

Macedo, H.D.,, Touili, T., 2013. Mining Malware Specifications Through Static Reachability Analysis, Lecture Notes in Comput. Sci. –ESORICS 2013 8134. Springer Berlin Heidelberg, 517–535.

McAfee Labs, 2016. McAfee Lab Threats Report: August 2015, McAfee Inc., [On-line]. Available electronically at⟨http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-aug-2015.pdf⟩.

Mehra, V., Jain, V., Uppal, D., 2015. DaCoMM: Detection and Classification of Metamorphic Malware. In: 5th International Conference on Commun. Syst. and Netw. Technol. (CSNT), IEEE, pp. 668–673.

Moser, A., Kruegel, C., Kirda, E., 2007. Limits of static analysis for malware detection. In ACSAC '07: Proceedings of the 23rd Annu. Comput. Secur. Appl. Conference ACSAC 2007, IEEE. Miami Beach, Miami Beach, FL, USA. pp. 421–430.

Park, Y., Reeves, D.S., Stamp, M., 2013. Deriving common malware behavior through graph clustering. J. Comput. Secur. 39, 419–430.

Piyanuntcharatsr, S.S.W., Adulkasem, S.,, Chantrapornchai, C., 2015. On the Comparison of malware detection methods using data mining with two feature sets. Int. J. Secur. Appl. 9 (3), 293–318.

Potier, J., 2016. WinAPIOverride32, [On-line].Available electronically at⟨http://jacquelin.potier.free.fr/winapioverride32/⟩

Ravi, C.,, Manoharan, R., 2012. Malware detection using windows API sequence and machine learning. Int. J. Comput. Appl. 43 (17), 12–16.

Rieck, K., Trinius, P., Willems, C.,, Holz, T., 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur. 19 (4), 639–668.

Salehi, Z., Sami, A.,, Ghiasi, M., 2014. Using feature generation from API calls for malware detection. J. Comput. Fraud Secur. 9, 9–18.

Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A., 2010. Malware detection based on mining API calls. In SAC '10: Proceedings of the 25th ACM Symp. on Applications Comput. ACM. Switzerland, pp. 1020–1025.

Shankarapani, M.K., Ramamoorthy, S., Movva, R.S.,, Mukkamala, S., 2011. Malware detection using assembly and API call sequences. J. Comput. Virol. 7 (2), 107–119.

Skaletsky, A., Devor, T., Chachmon, N., Cohn, R., Hazelwood, K., Vladimirov, V.,, Bach, M., 2010. Dynamic program analysis of Microsoft windows applications. perform. anal. of Syst. and Softw. (ISPASS). Int. Symp. IEEE. New Y., 2–12.

Sood, A.K., Enbody, R.J., 2013. Targeted cyber-attacks: a superset of advanced persistent threats. IEEE Secur. Priv. 11 (1), 54–61.

Szor, P., 2005. The Art of Computer Virus Research and DefenseAddison Wesley Prof. Symantec Press, Upper Saddle River, NJ.

Tahan, G., Rokach V., Shahar, Y., 2012. Mal-ID: Automatic malware detection using common segment analysis and meta-features. TheJ. Mach. Learn. Res. 13, 949–979.

Tian, R., Islam, R., Batten, L., Versteeg, S., 2010. Differentiating malware from clean ware using behavioral analysis. In: MALWARE '10: Proceedings of the 5th International Conference on Malicious and Unwanted Softw. Nancy, France. pp. 23–30.

Van Nhuong, N., Nhi, V. T. Y., Cam, N. T., Phu, M. X., Tan, C. D., 2014. Semantic set analysis for malware detection. In: Comput. Inf. Sys. Ind. Manag., Springer, Berlin, Heidelberg, vol. 8838, pp. 688–700.

Wagener, G., 2006. Development and design of a process and a piece of software to analyze unknown software. Univ. of Luxembourg, Luxembourg.

Walenstein, V., Hefner, D.J., Wichers, J., 2010. Header information in malware families and impact on automated classifiers. In: Malware '10: Proceedings of the 5th International Conference on Malicious and Unwanted Softw., Nancy, France. pp. 15–22.

Weng, C.G., Josiah, P., 2008. A new evaluation measure for imbalanced datasets. In: Proceedings of the 7th Australasian Data Mining Conference Australian Comput. Society. Glenelg, Australia, 87, 27–32.

Wüchner, T., Ochoa, M., Pretschner, A., 2014. Malware detection with quantitative data flow graphs, in Proceedings 9th ACM Symp. Inf. Comput. Commun. Secur. ACM. pp. 271–282.

Yason, M.V., 2007. The Art of Unpacking. MalcodeAnalyst, X-Force Research and Development IBM Internet Secur. Syst., Black Hat Briefings, USA.

Zamir, S., Margalit, Y., Margalit, D., 2004. Method for detecting unwanted executables. U.S. Patent Application10/890,170.

Zeng, J., Fu, Y., Miller, K. A., Lin, Z., Zhang, X., Xu, D., 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In: Proceedings of the 2013 ACM SIGSAC Conf. Comput. Commun. Secur. New Y. ACM, pp. 487–498. 2004.