Journal of Information Security and Applications 37 (2017) 91–100

Contents lists available at ScienceDirect

Journal of Information Security and Applications

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

Classification of malware families based on runtime behaviors

Abdurrahman Pektaş*, Tankut Acarman

Galatasaray University, Computer Enginnering Department, Ciragan Cad No:36, 34349, Ortakoy, Istanbul, Turkey

ARTICLE INFO

Article history:
Available online 3 November 2017

Keywords:
Behavior analysis
Dynamic analysis
Malware classification
Machine learning

ABSTRACT

Classification of malware samples plays a crucial role in building and maintaining security. Design of a malware classification system capable of supporting a large set of samples and adaptable to model changes at runtime is required to identify the high number of malware variants. In this paper, file system, network, registry activities observed during the execution traces and n-gram modeling over API-call sequences are used to represent behavior based features of a malware. We present a methodology to build the feature vector by using run-time behaviors by applying online machine learning algorithms for classification of malware samples in a distributed and scalable architecture. To validate the effectiveness and scalability, we evaluate our method on 17,900 recent malign codes such as viruses, trojans, backdoors, worms. Our experimental results show that the presented malware classification's training and testing accuracy is reached at 94% and 92.5%, respectively.

1. Introduction

Malicious software, or malware, is software used or created by an attacker to execute his/her bad intentions on a computer system without authorization and knowledge of its user. Basically, malicious attacks are targeted to steal safety-critical or liability-critical personal data or damage the compromised system. The recent developments in the field of computation system and proliferation of system such as smart phones, tablets, Internet of Things (IoT), cloud computing have led to an increased interest in malware development. The majority of new malware samples can be deployed as the variant of the previously known samples. The proliferation of the runtime packer and obfuscation techniques easily enables creation of behaviorally identical but statically different malware samples [1]. According to [2], more than 430 million new unique pieces of malware were detected in 2015 with an increase of 36% from the previous year, and a new zero-day vulnerability was discovered at each week on average with a doubled release frequency in comparison with the previous year.

Malware analysis can be grouped into two main categories based on whether or not the file under scrutiny is executed during the analysis: the associated methods can be referred as signature-based or behavior-based. Signature-based methods rely on unique raw byte patterns or regular expressions, known as signatures, created to match the malicious file. For instance, static features of a file are used to determine whether it is a benign or a malware. The main advantage of signature-based methods is their exhaustiveness since they trace all possible execution paths of a given file. Although these methods provide good detection rate on known samples, they are vulnerable to code obfuscation techniques such as run-time packing, metamorphism, and polymorphism that is generally used by malware authors to evade detection [3].

Unlike signature-based methods, behavior-based approaches require execution of a given sample in a sandboxed environment and run-time activities are monitored and logged. Dynamic analysis frameworks employ both virtualization and emulation environments to execute a malware and to extract its behaviors [4,5]. The behavior of an executable is extracted either by monitoring system changes made in the OS, or tracking API calls along with their parameters and returning values during execution. Although monitoring system changes is necessary to analyze behavior of a malware, this scheme does not involve monitoring some important behaviors (i.e., searching for specific file types or file name, enumeration of special registry keys, etc.,) adopted by advanced malware samples, for instance anti-VM technique to thwart analysis. Besides, some research efforts have focused on extracting behaviors based on the state changes between clean and dirty snapshots [6,7], where a clean (**dirty**) snapshot is a state of the machine before (**after**) execution of a malware sample.

Recently, for malware detection and determination whether being benign or malicious software, researchers have applied a fixed size n-gram and variable length n-gram that can be extracted from the binary content of the analysis file and opcodes obtained after dissembling [8,9]. For instance, a sequence of opcodes is used to

* Corresponding author.
E-mail addresses: apektas@yandex.com (A. Pektaş), tacarman@gsu.edu.tr (T. Acarman).

https://doi.org/10.1016/j.jisa.2017.10.005
2214-2126/

create a feature vector and three classifiers named as Ripper, C4.5 and IBK are used with ensemble learning algorithm to improve the accuracy in classification [10].

Meanwhile, function-call, control-flow, and data-flow graphs, which are more robust to code obfuscation than n-gram, are introduced for malware detection and classification [11]. In graph mining approaches, given software is simply presented as a graph. Then, this graph is compared with training graphs to identify the most similar one found in the dataset. Since graph matching is computationally expensive (an NP hard problem), graph comparing algorithms have been proposed to differentiate maliciousness from benign graphs.

Also, API calls reflect the aim of a program, and analyzing these calls can reveal the behavior of a program with less computational resource requirements. There are two methods to obtain the list of API calls: static analysis (e.g., IDA Pro-disassembly tool) or dynamic analysis (e.g., API hooking). Since a software can include multiple execution paths, dirty, and unused codes, extraction of API calls through static analysis by disassembler (e.g., with IDA Pro) is a challenging task. Moreover, disassemblers can be evaded by anti-disassembly methods. Last but not least, manual analysis of these calls can be a tedious task since a simple executable can make a considerably large amount of API calls. However, if a malware does not feature run-time protection, one can accurately obtain API calls through dynamic analysis.

In open literature, these methods are applied to detect and classify a malware by using different number of samples. The approach in [12] presents classification system based on a n-gram feature vector extracted from network level artifacts obtained via dynamic analysis. The evolution set of this work consists of 3 families and includes around 3000 samples. By using SVN, k-NN and decision tree, 80% accuracy in classification is achieved. In [13], the API calls and their arguments are used to model behavior. But evaluation is made with a limited amount of malware samples. A pre-defined set of API calls and a narrow feature space built to represent a software is used in [14] but crucial information about behavior is not extracted due to poor modeling of a malware. In [15], a behavioral fingerprint of a malware is composed of system state changes such as files written, processes created and rather than sequences or patterns of system calls. To measure similarity among the malware groups, a tree structure based on single-linkage clustering algorithm is presented. The method is tested by using real world malware samples (including samples that have not been detected yet, and therefore do not have a signature) and more successful classification results are obtained in comparison with anti-viruses using signature-based methods.

In [16], a classification method is introduced in order to determine whether a given malware sample is a variant of known malware family or a new malware strain. System call traces are captured and the behavior of malicious software is monitored by means of special representation called Malware Instruction Set (MIST), which is inspired from instruction sets used in CPU. In this representation, the behavior of a sample is characterized with a sequence of instructions. A behavior-based automated classification method, which is motivated by [16], is proposed in [17]. Dynamic analysis report gives the status change caused by the executable and events, this information is obtained from corresponding Win32 API calls and their certain parameters. Behavior unit strings are extracted as the features in order to distinguish malware families. To reduce the dimension of feature space, string similarity and information gain measure is used. A malware classification method using runtime actions and API calls of malware samples is presented in [18]. Supervised machine learning Random Forests is applied with 160 trees to classify 42,000 malware samples into 4 malware families. True positive rate is reached at 0.896 and false positive rate at 0.049 subject to the restricted number of families. In [19], malware samples are detected first and then classified as unknown or known malware by applying Random Forests classifier. Behavioral traces and API calls along with input parameters are used to build the feature vector. 31,295 malware samples belonging to 5 families and 837 benign samples are used for determining whether they are known or unknown, true positive rate and false positive rate is reached at 0.981and 0.099 subject to the 5 respective malware families. In [20], malware detection based on API call sequence analysis is presented. Malware samples are executed in a virtual environment and API call sequences are traced during runtime by using user-space hooking library called Detour [21]. Then, DNA sequence alignment technique is applied to remove meaningless codes inserted into malware samples. Finally, the common API call sequence patterns among malware are extracted by applying the longest common sub-sequences (LCS) algorithm. 2727 kinds of API into 26 groups are categorized in accordance with MSDN library. Classification accuracy is reached at 99% as the result of testing dataset consisting of 6910 malware and 34 benign samples. The main limitation of this method is that computing LCS and DNA sequence alignment is NP-hard problem, therefore computational complexity is high requiring more computational resources and time.

Throughout evaluation of a malware and application of online machine learning algorithms, a trade-off exists between the scalability of large-scale malware classification and computational complexity. For instance, when the feature space increases, data become sparse and the computation time of algorithms increases exponentially with the number of malware samples making the analysis inefficient. This problem is also known as the *curse of dimensionality*.

In this paper, we present a malware classification methodology while grouping samples based on their runtime behavior patterns by applying online machine learning. We capture implicit features of behavior in order to improve the accuracy of classifying malware. We perform an extensive assessment of our technique using standard classification evaluation metrics (e.g., accuracy, precision, recall, F1-score) and a large number of malware families, showing favorable evaluation results. Furthermore, we present the computational resource usages needed to deploy the presented classification methodology. A preliminary version of this study was presented in [22] and run-time behaviors were extracted to build the feature vector. Compared to [22], additional results about using API call sequences, resource usage with a more complete background are elaborated.

The rest of the paper is organized as follows: Section 2 describes the methodology for extracting behavior of the file under analysis along with the implementation details. Experiments and their results are discussed in Section 3. Conclusions and limitations are given in Section 4.

2. Methodology and implementation

The proposed framework consists of three major stages. The first stage consists of extracting the behavior of the sample file under scrutiny and observing its interactions with the OS resources. At this stage, the sample file is run in two sandboxed environment; VirMon [4] and Cuckoo [5]. During the second stage, we apply feature extraction to the analysis report. The label of each sample is determined by using Virustotal [23]. Then at the final stage, the malware dataset is partitioned into training and testing set. The training set is used to obtain a classification model and the testing set is used for evaluation purposes. An overview of our system including its main functionalities is presented in Fig. 1.

From the viewpoint of this study, the run-time behavior of a given file is modeled by fusing both API calls and changes made

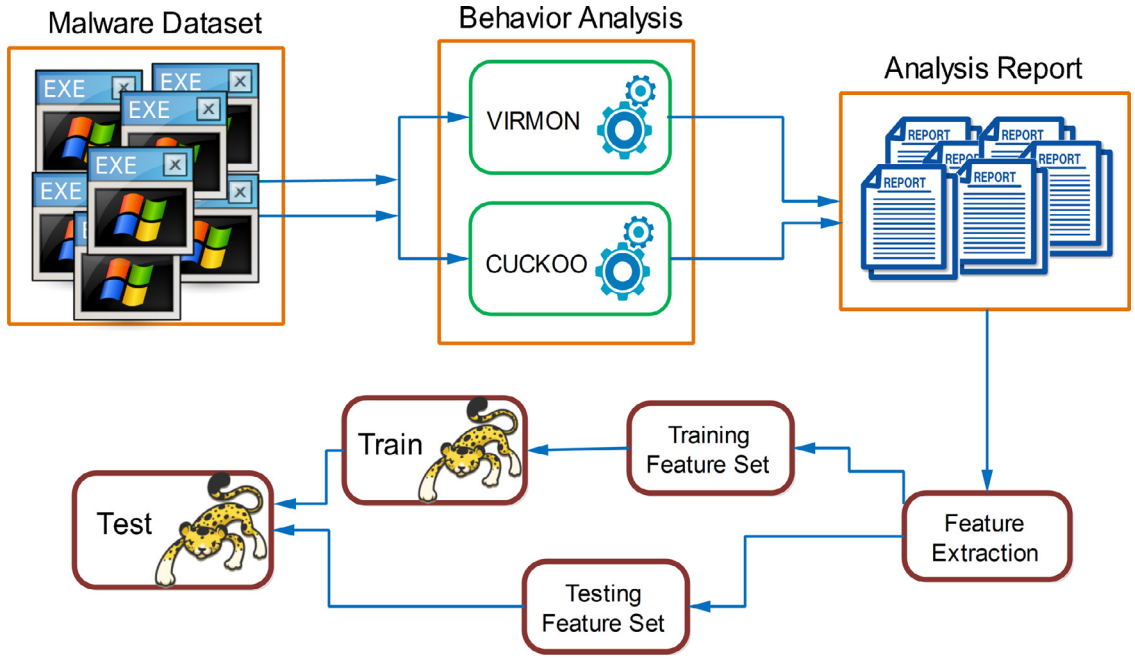


Fig. 1. Overview of the proposed malware classification system.

Table 1
Adopted features from dynamic analysis frameworks.

	Mutexes	Process tree	IDS signatures	DNS requests	HTTP requests	File activities	Registry activities	Service activities	IRC commands	API calls
Cuckoo	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔
VirMon	✘	✔	✔	✔	✔	✔	✔	✘	✔	✘

to the OS. To extract the knowledge about the run-time behavior of a file, two underlying automated dynamic platforms are used: VirMon is deployed for extracting Windows kernel-level notification routines [4] and Cuckoo is customized for scalability [5]. The adopted features of VirMon and Cuckoo to extract behaviors are listed in Table 1.

In VirMon, any system change occurring on the analysis machine is monitored through Windows kernel-level notification routines. During the analysis, the state changes of the OS resources such as file, registry, process/thread, network activities and IDS alerts are logged into a report file. But VirMon does not extract API call sequence; open source Cuckoo sandbox is used to provide function calls of the file under analysis through hooking method. Cuckoo reports artifacts with its agents while executing a file in an isolated environment.

We modified default Cuckoo configuration in order to increase the number of concurrent analysis. The scalability of the analysis system is assured by dedicating adequate computers for the expected analysis work. The number of guest machines in one analysis server is configured depending on the host system used for analysis. For example, 10 guest machines run in a server containing 32 GB memory and 4×4 Xeon CPUs. We selected Windows XP SP3 as analysis OS since it consumes less memory and CPU power. Five host machines with the same configuration are employed. On each of these host machines, 10 guest machines are dedicated to dynamically analyze malware samples. Overall, Cuckoo was configured to have the total of 50 analysis machines. The curl utility is used to automatically submit files into Cuckoo. Although, the analysis report can be saved into either relational or non-relation database, we prefer to save into file system. A storage unit is formatted as glusterfs [24] file format and analysis reports are shared between host machines over the network.

2.1. Malware classification framework & feature extraction

The problem faced with classification of the malware using machine learning is that the feature space is very large. Researchers have proposed many techniques to reduce the feature space [15–17]. However, the proposed approaches do not scale well to large datasets and require extensive computational resources. Distributed machine learning frameworks have been proposed. These frameworks can be separated into two groups according to mechanism for updating the model: online frameworks that can update their model subject to training samples and batch (or offline) frameworks that can update the used models on a periodical basis.

An online framework is more useful to properly separate malware samples into their respective classes. The requirement about an immediate model update subject to the inclusion of a new malware detected in the wild is satisfied. We have chosen Jubatus [25], an online machine learning framework, to classify malware samples based on their behavioral patterns. We set up a cluster of five machines for Jubatus that allows us not only to speed up the classification process but also to process a large scale of dataset.

There is no standard method for sharing information regarding the presence of malware on the computer system. Frameworks such as Open Indicators of Compromise (OpenIOC) [26] and Malware Attribute Enumeration and Characterization (MAEC) [27] have been introduced to identify malware based on its network and host level indicators instead of hash values or signatures employed by the conventional security tools. Consequently, these standardized malware reporting formats provide an opportunity to characterize malware samples uniformly and to prevent malware community from redundant analysis. In this study, common malicious features, which are also the most significant and representative,

Table 2
Features and their types.

Feature category	*Type*	*Value*
Sequence of API category	N-gram	'ABCA', 'BACA', ...
Mutex names	String	'z3sd'
Created processes	String	'reg.exe'
Copy itself	Boolean	False
Delete itself	Boolean	False
DNS requests	String	Remote IPs
Remote IPs	String	'54.209.61.132216.38.220.26'
TCP Dst port	String	'80'
UDP Dst port	String	*None*
Presence of the special APIs	Boolean	'isdebuggerpresent': False, 'setwindowshook': True
Read files	String	*None*
Registry keys	String	*None*
Changed/created files	String	'%Document and Settings%/ftpdll.dll %SYSTEM%/drivers/spools.exe %SYSTEM%/ftpdll.dll %APP_DATA%/cftmon.exe... '
Changed/created registry keys	String	'HKCU\Software\Microsoft\Windows\CurrentVersion\Run HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon... '
Downloaded EXE	String	'spools.exe cftmon.exe'
Downloaded DLLs	String	'ftpdll.dll'
User-agents	String	'_'
IDS signature	String	*None*
HTTP requests	String	'/?&v=Chatty/domain_profile.cfm?d=fewfwe&e=com'
ICMP data	String	*None*
ICMP host	String	*None*
IRC commands	String	*None*

are used to present malware behavior in compliance with both MAEC and OpenIOC. Feature extraction is applied to the analysis report obtained from the dynamic analysis. We also know that a malware tends to utilize random names to complicate malware analysis. Runtime features are sanitized from random values that are changed by malware before each run. Table 2 shows the set of features and the variable types for the file with MD5 value a27d774a8ce7846dfd5ae40d4411cb81.

2.2. N-gram modeling over API-call sequences

In the field of statistic, an n-gram is a fixed size sliding window of byte array, where n is the size of the window. For example, the '81EDD871' sequence is segmented (represented) into 5-gram as '81EDD', '1EDD8', 'EDD87' and 'DD871'. Essentially, n-gram modeling is used to build a feature vector for the given sequences of many different samples and prediction is made by comparing extracted n-gram features of a given sample file versus this feature vector.

As malware authors must execute relevant API calls to achieve their malicious goals, analysis of API calls will allow malware examiner understand the behavior of a given sample. Hence, the sequence of Windows API calls can be used to model the behavior of an executable file. The Windows API function calls can be grouped into various functional categories such as system, registry, services and network activities. In this study instead of API call based modeling, we use category based modeling, which leads to a lower feature space. We encode API calls used by a malware during dynamic analysis into one length long codes (i.e., characters), by ignoring the successive API calls, the encoding method effectively captures the semantics of the API calls while being resilient to obfuscation techniques. Actually, 15 different categories of API calls are utilized to build n-gram features, categorized API calls with their numbers and encoding characters are listed in Table 3. Categorization of API calls reduces the feature space, and processing of the machine learning algorithm requires less time and CPU power.

To assign a weight to each feature obtained from n-gram extraction, the metric of frequency and inverse document frequency is used. The statistical measure about how often a term occurs in a document is the frequency metric, denoted by **tf**. Inverse document frequency, denoted by **idf**, measures whether the given term is common or rare in all documents (in our case, a document refers to a malware sample). **idf** leverages the feature that rarely occurs in the document corpus. Ultimately, the weight of each feature is the product of these two metrics.

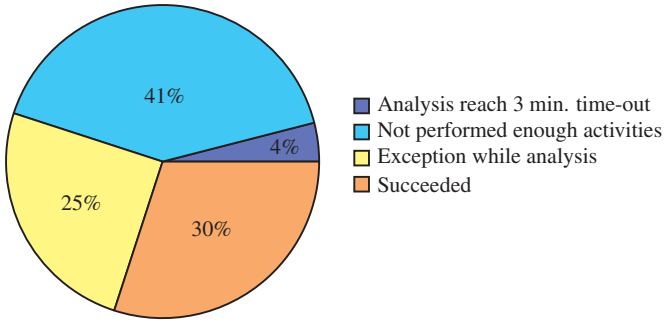


Fig. 2. Results of dynamic analysis about the evaluation set.

3. Experiments and results

3.1. The malware dataset

The testing malware dataset was obtained from "VirusShare Malware Sharing Platform" [28]. A large amount of malware with different types including PE, HTML, Flash, Java, PDF, APK was downloaded. Then, since VirMon can only analyze executable files, executable files are considered. To analyze actual malware trends, the samples collected by VirusShare are chosen for behavioral analysis. Throughout the analysis, 25% of samples belonging to this dataset did not run because either some samples required user interaction, some particular samples checked hardware specification or some samples could be deployed on a version of .NET not provided in Windows XP SP3. The responses of samples to the analysis and their ratios are plotted in Fig. 2. At the end of analysis, 45,000 files were correctly analyzed and reported. However, more

Table 3
API calls and their categories.

Category	Code	API #	APIs
Hooking	A	1	unhookwindowshookex
Network	B	19	dnsquery_a, dnsquery_utf8, getaddrinfo, getaddrinfow, httpopenrequesta, ...
Windows	C	4	findwindowa, findwindoww, findwindowexa, findwindowexw
Process	D	21	createprocessinternalw, exitprocess, ntallocatevirtualmemory, ntcreateprocess, ...
Misc	E	2	getcursorpos, getsystemmetrics
System	F	12	exitwindowsex, isdebuggerpresent, ldrgetdllhandle, ldrgetprocedureaddress, ...
Threading	G	12	createremotethread, createthread, exitthread, ntgetcontextthread, ntcreatethreadex,...
Synchronization	H	3	ntcreatemutant, ntcreatenamedpipefile, ntopenmutant
Device	I	1	deviceiocontrol
Registry	J	38	ntcreatekey, ntdeletekey, ntdeletevaluekey, ntenumeratekey, ntenumeratevaluekey, ...
Filesystem	K	23	createdirectoryw, createdirectoryexw, removedirectorya, removedirectoryw, ...
Services	L	10	controlservice, createservicea, createservicew, deleteservice, openscmanagera, ...
Socket	M	24	accept, bind, closesocket, connect, connectex, gethostbyname, ...

Table 4
Classification accuracy of online learning algorithms versus different regularization weight parameter.

Algorithm	Regularization weight(C)									
	C=1.0		C=2.0		**C=3.0**		C=4.0		C=5.0	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
PA-I	0.948	0.454	0.945	0.45	0.94	0.438	0.939	0.431	0.935	0.453
PA-II	0.947	0.433	0.944	0.485	0.935	0.418	0.935	0.465	0.93	0.448
CW	0.946	0.863	0.947	0.870	**0.947**	**0.871**	0.943	0.863	0.941	0.853
AROW	0.947	0.773	0.940	0.750	0.944	0.718	0.939	0.754	0.935	0.756
NHERD	0.927	0.767	0.853	0.667	0.846	0.656	0.844	0.681	0.827	0.539

than half of these correctly analyzed samples did not illustrate enough activities. Since these samples could cause false positives, we removed them from the dataset. All the experiments were conducted under the Windows XP SP 3 operating system with Intel(R) Core(TM) i5-2410M@2.30 GHz processor and 1GB of RAM. Overall, the analysis with 50 guest machines took 15 days to analyze 60,000 samples and prepare a set of testing dataset constituted by 17,900 responding samples.

For labeling a malware sample, we used Virustotal [23] as an online web-based multi anti-virus scanner. Since malware labeling is not unique and may differ between different anti-virus engines, labeling process can affect the classification accuracy. Therefore, researchers need to have a priori knowledge about anti-virus labeling and need to cross check the labeling results. In this study, the most common scan result is determined as the tag of a sample in order to assure consistent labeling. The set of malware samples is randomly divided into 10 equal size subsets and 10-fold cross-validation approach is used. A single subset is used for testing the model and the remaining 9 subsets are used for building the classification model. This process is then repeated 10 times such that each subset is used at lest once as the testing data. Finally, the classification accuracy is calculated by averaging the accuracy obtained at the end of 10 iterations.

3.2. Online classification algorithm

In general, an online learning algorithm works in a sequence of consecutive rounds. At round t, the algorithm considers an instance $\vec{x}_t \in \mathbb{R}^d$, d-dimensional vector, as an input to make the prediction $\hat{y}_t \in \{+1, -1\}$ (for binary classification) regarding to its current prediction model. After predicting, it receives the true label $y_t \in \{+1, -1\}$ and updates its model (a.k.a. hypothesis) based on prediction loss $\ell(y_t, \hat{y}_t)$ meaning the incompatibility between prediction and actual class. The goal of online learning is to minimize the total number of incorrect predictions; $sum(t : y_t \neq \hat{y}_t)$. Pseudo-code for generic online learning is given in Algorithm-1.

Multi-class classification problem is considered:

- $\vec{x}_t$ represents the feature vector of a malware instance at the tth iteration. When implementing the algorithm for detection of a malware sample or a malware's variant, its set of features is constituted by using the basis of independent feature vector set given in Table 2. For each independent feature, its category, type and value is known so the sample can be represented in terms of these independent features.
- $\vec{y}_t$ is the set of labels at the tth iteration. $\hat{y}_t$ is the output of the algorithm or more simply, prediction of malware family for the given $\vec{x}_t$.
- ℓ_t is the function using the relation between the set of features for the given sample, the computed weight, denoted by $\vec{w}_t$ and the estimated malware label.
- $\vec{w}_{t+1}$ denotes the updated weight vector at the $(t+1)$th iteration towards the final class prediction.

Our focus in particular is not on theoretical contribution to online learning as this has been addressed in many studies over the past years. Instead, the attention is drawn to a scalable and reliable implementation of malware classification. Online machine learning algorithms differ according to how to initialize the weight vector $\vec{w}_{t=1}$ and update function used to alter the weight vector at the end of each iteration. The following online classification algorithms are used in our distributed computing environment in order to empirically reach at the highest level in accuracy, (the interested readers may investigate the references).

- Passive-Aggressive I (PA-I) [29]
- Passive-Aggressive II (PA-II) [29]
- Confidence Weighted Learning (CW) [30]
- Adaptive Regularization of Weight (AROW) [31]
- Normal Herd (NHERD) [32]

3.3. Classification metrics

To evaluate the proposed method, the following class-specific metrics are used: ***precision, recall*** (a.k.a. sensitivity), ***specificity, balanced accuracy***, and ***overall accuracy*** (the overall correctness of the model). Recall is the probability for a sample in class c to be

Table 5
Training and testing accuracy of CW.

	Regularization weight(C)													
	$C = 1.0$		$C = 2.0$		$C = 3.0$		***C*=4.0**		$C = 5.0$		$C = 10.0$		$C = 100.0$	
n	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
3	0.890	0.886	0.893	0.884	0.800	0.828	0.690	0.768	0.617	0.616	0.340	0.33	0.083	0.186
4	0.907	0.894	0.920	0.898	0.910	0.900	0.917	0.894	0.913	0.896	0.883	0.874	0.733	0.632
5	0.907	0.900	0.913	0.902	0.923	0.904	0.930	0.906	0.927	0.904	0.930	0.906	0.877	0.816
6	0.917	0.896	0.920	0.908	0.927	0.908	**0.940**	**0.918**	0.937	0.908	0.937	0.904	0.883	0.826
7	0.917	0.900	0.930	0.910	0.930	0.912	0.930	0.910	0.930	0.910	0.933	0.906	0.500	0.824
8	0.910	0.900	0.923	0.904	0.933	0.908	0.933	0.908	0.930	0.904	0.937	0.908	0.853	0.866
9	0.913	0.898	0.923	0.902	0.923	0.902	0.920	0.904	0.927	0.906	0.930	0.900	0.907	0.882
10	0.913	0.718	0.930	0.898	0.930	0.902	0.920	0.898	0.927	0.904	0.920	0.898	0.913	0.882

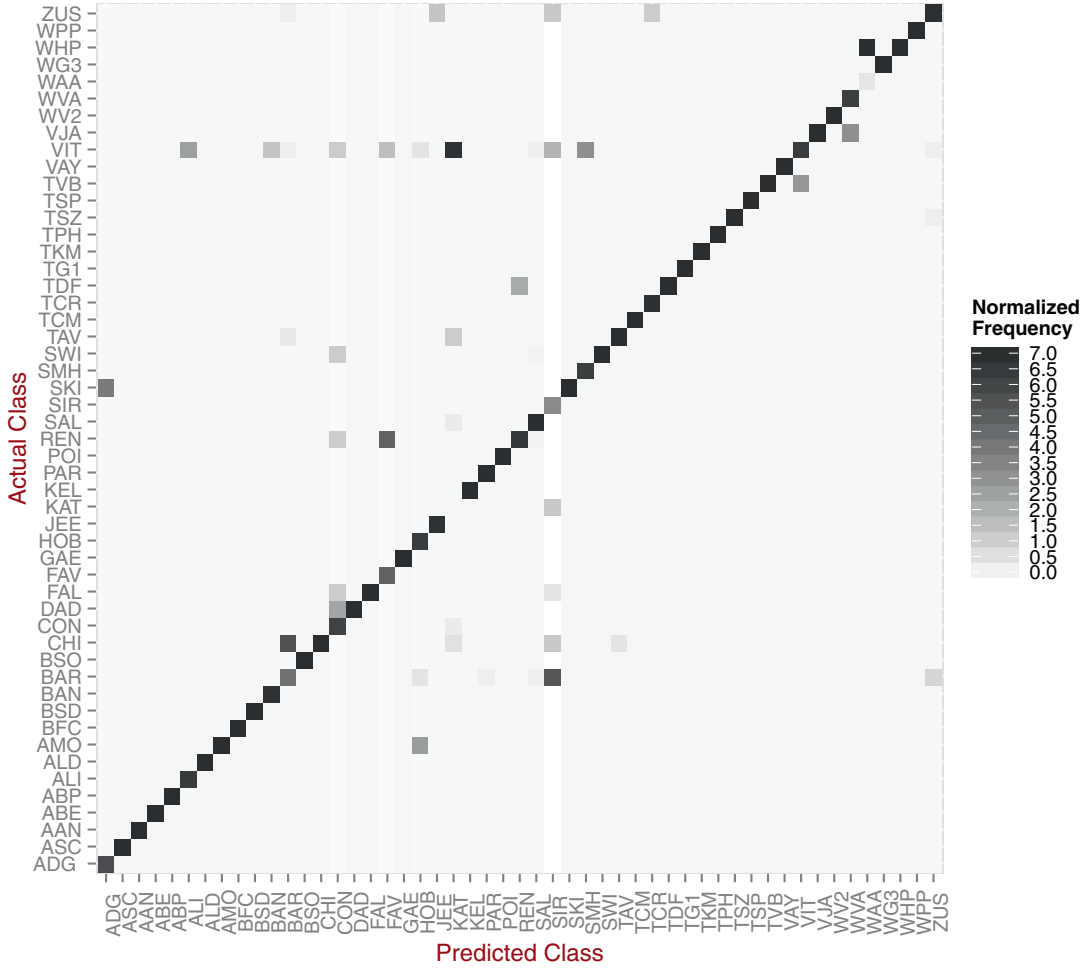


Fig. 3. Normalized confusion matrix.

classified correctly, the maximum value 1 means that the classifier is always correct about its prediction whether an instance belongs to class c. On the contrary, specificity is the probability for a sample not in class c to be classified correctly, the maximum value 1 means that the classifier is always correct about an instance that does not belong to class c. Precision gives the probability for an estimated instance about classification in class c to be actually in class c. Low precision means that a large number of samples are incorrectly classified in class c.

Balanced accuracy is a class-specific performance metric used to evaluate overall misleading accuracy when the dataset is imbalanced. More specifically, this metric assesses the classification error for each particular class and error is calculated as the arithmetic mean of the specificity and recall. Overall accuracy is calculated as the ratio of correct classifications versus the total number of samples. The metrics are given as follows:

$$precision = \frac{tp}{tp + fp} \tag{1}$$

$$recall = \frac{tp}{tp + fn} \tag{2}$$

$$specificity = \frac{tn}{tn + fp} \tag{3}$$

$$balanced\ accuracy = \frac{recall + specificity}{2} = \frac{1}{2}\left(\frac{tp}{tp + fn} + \frac{tn}{tn + fp}\right) \tag{4}$$

Table 6
Malware families and class-specific performance measures.

Family	*Code*	*#*	*Precision*	*Recall*	*Specificity*	*Balanced accuracy*
AdGazelle	ADG	121	0.59	1.00	0.99	1.00
Adw.ScreenBlaze	ASC	40	1.00	1.00	1.00	1.00
Adware.Agent.NZS	AAN	28	1.00	1.00	1.00	1.00
Adware.BetterSurf	ABE	39	1.00	1.00	1.00	1.00
Adware.Bprotector	ABP	91	1.00	1.00	1.00	1.00
Aliser	ALI	48	0.71	1.00	1.00	1.00
Almanahe.D	ALD	16	1.00	1.00	1.00	1.00
Amonetize	AMO	67	1.00	0.29	1.00	0.64
Backdoor.Fynloski.C	BFC	42	1.00	1.00	1.00	1.00
Backdoor.SpyBot.DMW	BSD	20	1.00	1.00	1.00	1.00
Banker	BAN	49	0.83	1.00	1.00	1.00
Barys	BAR	403	0.40	0.50	0.98	0.74
Bundler.Somoto	BSO	982	1.00	1.00	1.00	1.00
Chinky	CHI	433	1.00	0.30	1.00	0.65
Conjar	CON	59	0.45	0.83	1.00	0.91
Dialer.Adultbrowser	DAD	25	1.00	0.33	1.00	0.67
FakeAlert	FAL	25	1.00	0.33	1.00	0.67
FakeAV	FAV	38	0.43	1.00	1.00	1.00
Gael	GAE	10	1.00	1.00	1.00	1.00
Hotbar	HOB	126	0.63	1.00	1.00	1.00
Jeefo	JEE	49	0.83	1.00	1.00	1.00
Kates	KAT	18	0.00	0.00	0.98	0.49
Keylog	KEL	21	1.00	1.00	1.00	1.00
Parite	PAR	370	0.97	1.00	1.00	1.00
PoisonIvy	POI	38	1.00	1.00	1.00	1.00
Renos	REN	116	0.75	0.55	1.00	0.77
Sality	SAL	604	0.91	0.98	1.00	0.99
Sirefef	SIR	51	0.22	1.00	0.99	0.99
Skintrim	SKI	202	1.00	0.55	1.00	0.78
SMSHoax	SMH	43	0.67	1.00	1.00	1.00
Swizzor	SWI	536	1.00	0.96	1.00	0.98
Trojan.Agent.VB	TAV	197	0.93	0.70	1.00	0.85
Trojan.Clicker.MWU	TCM	17	1.00	1.00	1.00	1.00
Trojan.Crypt	TCR	58	0.86	1.00	1.00	1.00
Trojan.Downloader.FakeAV	TDF	16	1.00	0.33	1.00	0.67
Trojan.Generic.1733394	TG1	2507	1.00	1.00	1.00	1.00
Trojan.Keylogger.MWQ	TKM	21	1.00	1.00	1.00	1.00
Trojan.Patched.HE	TPH	28	1.00	1.00	1.00	1.00
Trojan.Startpage.ZQR	TSZ	23	1.00	0.33	1.00	0.67
Trojan.Stpage	TSP	136	1.00	1.00	1.00	1.00
Trojan.VB.Bugsban.A	TVB	199	1.00	0.15	1.00	0.58
Variant.Application.Yek	VAY	18	1.00	1.00	1.00	1.00
Virtob	VIT	763	0.69	0.48	0.99	0.74
VJadtre	VJA	134	1.00	0.69	1.00	0.85
Win32.Valhalla.204	WV2	20	1.00	1.00	1.00	1.00
Win32.Viking.AU	WVA	77	0.67	1.00	1.00	1.00
Worm.AutoIt	WAA	53	0.06	1.00	0.94	0.97
Worm.Generic.384701	WG3	7231	1.00	1.00	1.00	1.00
Worm.Hybris.PLI	WHP	998	1.00	0.01	1.00	0.50
Worm.P2P.Palevo	WPP	24	1.00	1.00	1.00	1.00
Zusy	ZUS	670	0.84	0.93	0.99	0.96

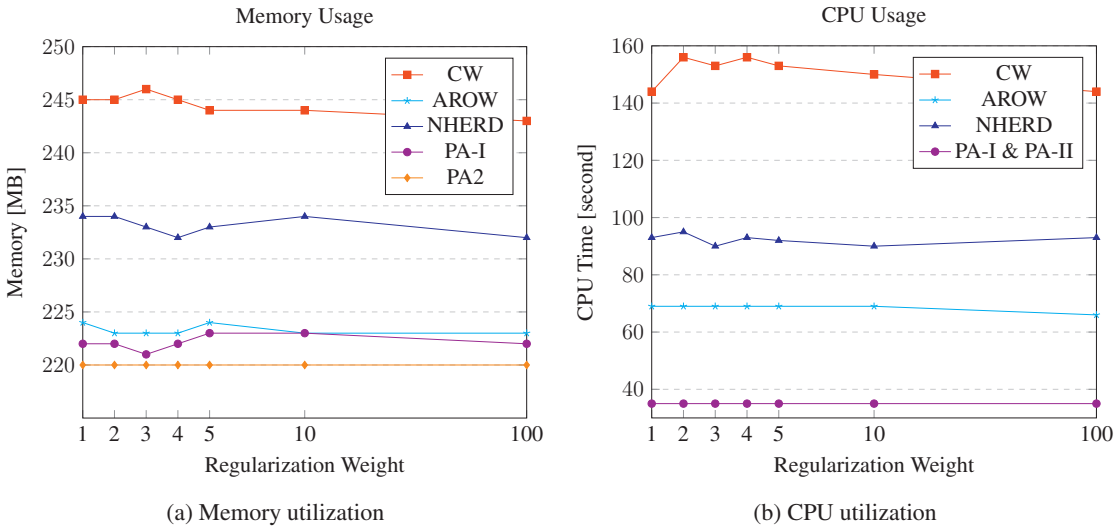
$$accuracy = \frac{correctly\ classified\ instances}{total\ number\ of\ instances} \tag{5}$$

For instance, consider a given class c. True positives (tp) refer to the number of the samples in class c that are correctly classified while true negatives (tn) are the number of the samples not in class c that are correctly classified. False positives (fp) refer the number of the samples not in class c that are incorrectly classified. Similarly, false negatives (fn) are the number of the samples in class c that are incorrectly classified. The term positive and negative indicates the classifier's success, and the term true and false denotes whether or not that prediction is matched with the ground truth label.

3.4. Training and testing accuracy and classification results

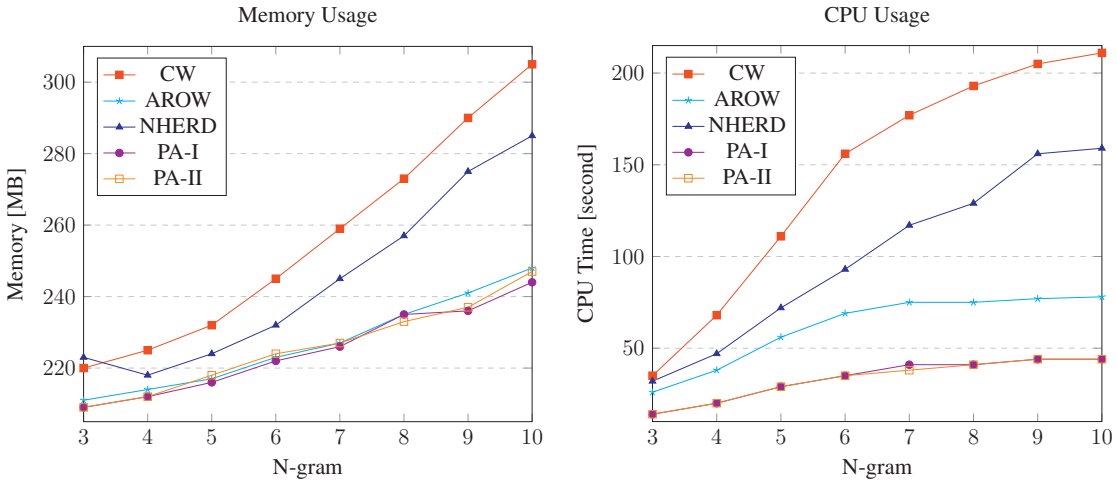
The accuracy of training and testing for each online learning algorithm is computed subject to different value of regularization weight parameter. The accuracy results in Table 4 are obtained by using the features listed in Table 2 except the Sequence of API category for the n-gram type (the first line in Table 2) is not used. Although high training accuracy can be reached at the training phase, such as 94% training accuracy for almost each online learning algorithm, testing accuracy is poor for PA-I, PA-II and NHERD algorithm in comparison with CW and AROW algorithm. Training and testing accuracy varies greatly due to model building capability while using the same set of features and samples.

We are also motivated by the knowledge that Windows API call sequence is not changed by obfuscation techniques, and the API calls encoded into the characters can be used to analyze behavior of an executable file and to enhance the classification accuracy. Hence, the length denoted by n is a design parameter to be tuned for increasing testing accuracy. The features listed in Table 2 including the Sequence of API category with a type named as n-gram is used. In Table 5, the most accurate classification results for training and testing are obtained by applying Confidence



(a) Memory utilization (b) CPU utilization

Fig. 4. The performance of the selected algorithms based on regularization weight when $n = 6$.



(a) Memory utilization (b) CPU utilization

Fig. 5. The performance of the selected algorithms based on n-gram when $C = 4.0$.

Algorithm 1 Generic online machine learning algorithm.

Initialize: $\vec{w}_{t=1} = (0, \ldots, 0)$
  for each round t in (1,2,..,N) **do**
    Receive instance $\vec{x}_t \in \mathbb{R}^d$
    Predict label of $\vec{x}_t$: $\hat{y}_t = sign(\vec{x}_t.\vec{w}_t)$
    Obtain true label of the $\vec{x}_t$: $y_t \in \{+1, -1\}$
    Calculate the loss: ℓ_t
    Update the weights: $\vec{w}_{t+1}$
  end for
Output: $\vec{w}_{t=N} = (w_1, \ldots, w_d)$

Weighted Learning (CW) algorithm when the regularization weight C=4.0 and window size n=6. The malware classes, or namely families, used to evaluate the proposed method and the class-wise metric results for CW algorithm are given in Table 6. These results indicate that perfect precision and recall value (i.e., 1.0) is achieved for 20 out of 51 families. For example, Adw.ScreenBlaze, Worm.Generic.384701 and Gael are one of these families. The classifier predicts them without error. Worm.Hybris.PLI family exhibits perfect precision but low recall, indicating that the classifier inaccurately predicts almost all instances. For Kates family, the classifier achieves both zero precision and recall, which indicates that it never correctly classified an instance belonging to Kates family, in other words *tp* is 0. It is important to note that the classifier predicts the Worm.Generic.384701, which has the highest number of samples and covers almost 40% of dataset without error.

The confusion matrix illustrates the success of the classifier at recognizing instance of different classes in Fig. 3. The confusion matrix displays the number of correct and incorrect predictions made by the classifier with respect to ground truth (correct classes). The confusion matrix has $m \times m$ entries, where m is the total number of independent classes. The rows of the table correspond to actual classes and columns correspond to predicted classes (or namely, families) as listed in Table 6. The diagonal elements in the matrix represent the number of correctly classified instances for each class, while the off-diagonal elements represent the number of incorrectly classified elements.

The confusion matrix illustrates that Worm.Hybris.PLI and Virtob is wrongly predicted as Worm.Autoit and Kates, respectively. A quick search on the Internet provides the information that some AV vendors give Worm.Autoit label instead of one used in our dataset (i.e., Worm.Hybris.PLI). Interested readers can find justify-

Table 7
Comparison of proposed malware classification method with current studies.

Study, Year	*Algorithm*	*Features*	*Type*	*Dataset*	*Accuracy*
[12], 2014	SVM	N-gram feature of the network artifacts	Classification	3000 samples, 3 families	80%
[14], 2013	SVM, LR	Set of OS actions	Detection	5300 samples, 100 benign	99%
[15], 2007	Single-linkage hierarchical clustering using normalized compress distance	Run-time artifacts	Clustering	3700 samples	91%
[16], 2008	SVM	System call traces	Classification	10,072 samples, 14 families	80%
[18], 2015	Supervised learning Random Forests	System call traces and APIs	Classification	42,000 samples, 4 families	tpr: 0,9 fpr: 0,05 (precision: 0,94)
[19], 2016	Supervised learning Random Forests	System call traces and APIs	Classification	31,295 samples, 837 benign, 5 families	tpr: 0,98 fpr: 0,01 (precision: 0,90)
[20], 2015	DNS sequence alignment	API Calls	Classification	6910 samples, 34 benign	99%
[35], 2014	Ensemble learning with Ripper, C4.5 and IBk	N-gram feature of the disassembled code	Detection	Unknown	Unknown
[36], 2012	Information Gain & Adaboost with base classifiers	API calls and their parameters	Classification	1368 samples, 10 families	97%
[37], 2010	AdaboostM1 with 5 base classifiers; SVM, perceptron,etc.	Function length frequency and printable string information	Detection	1400 unpacked malware, 151 benign	98%
[38], 2012	One-class SVM	APIs, strings and basic blocks	Classification	113 samples	78%
[39], 2013	Online machine learning (algorithm unknown)	Performance monitor, system call and system call sequences	Classification	3454 samples, 32 families	66.8%
[40], 2013	Online machine learning (CW, ARROW)	Features derived from URL string	Detection	1,000,000 URLs	75%
Our study, 2016	Online machine learning (CW, ARROW, PA-I & II, NHERD)	Runtime artifacts, IDS signatures, important API calls	Classification	17,900 samples, 51 families	92.5%, (average precision: 0,86)

ing examples about wrong classification and mislabeling in [33]. Overall, accuracy level in testing is reached at 92.5%. We analyzed the run-time behavior Virtob and realized that some of its samples generate almost similar artifacts with Kates, such as modifying same registry keys related to Internet Explorer settings and auto-start location, and also using same mutex names. The reports about behavior of these two samples can be followed by providing their MD5 values af5ce0870db09f5f9cd9ab2bd62f42ef and 91cad3f61b7898a0a9969c1ad46883a4 in [34].

We evaluate the responsiveness of the proposed classification method by using a new malware sample that has not been trained and tested by the online learning system yet, for instance, when the run-time model needs to be updated. In order to accomplish this new malware evaluation task, we gather 101 new malware samples belonging to the same family -Win32/CeeInject- from VirusShare. After obtaining behavioral profiles from dynamic analysis, only one sample is randomly selected from this new sample set, and it is used to train and update the model. Subsequently, the remaining 100 samples are evaluated by the updated model during 1.3 seconds. The results show that all samples are classified correctly and the tested samples are labeled as Win32/CeeInject. It should be noted that the proposed method does not require training again by using the large dataset to update the run-time model.

Fig. 4 shows the CPU and memory usage of the selected online learning algorithms according to various regularization weight (C) values when n-gram size is kept constant at $n = 6$ Fig. 5 shows the CPU and memory utilization according to various n-gram size when the regularization weight is constant at $C = 6$. The performance measures are collected from a single Jubatus server via *ps* command. Without loosing of generality, the memory and CPU usage is enforced by computation of the weight vector and computational complexity of its update. By following their references, weight update of the PA-I and PA-II algorithm is a classical constrained optimization problem and it is linearly dependent on non-negative scalar variable preventing change in the wrong direction due to mislabeled samples, [29]. The AROW, NHERD and CW algorithm assumes a Gaussian distribution over weight vectors. AROW updates mean vector, covariance matrix and a regularization parameter is added at consecutive learning iterations, [31].

The weight vector update whether the weight distribution matrix is required influences computational resource usage. In Fig. 4, usage of memory and CPU for PA-I and PA-II is comparably lower versus other algorithms and it is the highest for the CW algorithm. Then, NHERD requires updating the matrix subject to a linear transformation and its resource requirement is ranked in the middle versus the implemented algorithms. AROW, PA-I and PA-II uses less resources in order to update weight vector but AROW calculates a mean vector and a covariance matrix requiring more CPU cycles. Overall, when the regularization weight, denoted by C, used to update the model is chosen low, the model creation takes more time and requires more resources on a CPU and memory. However, when this weight is chosen high, model building is faster and requires less resources.

Based on the graphics illustrated in Fig. 4, the CPU and memory usage varies versus the parameter C, CPU and memory usage is reached at its maximum versus other resource usages when $C = 2$ and $C = 3$. The model is built slower versus higher regularization weight values. Higher regularization weights are ($C = [5, 10, 100]$) and they allow faster model building. In Fig. 5, the resource usage is plotted versus different size of n-gram. Since n defines the number of API calls encoded into the characters, the memory is used linearly to the number of n and CPU resource is used exponentially versus the increase in n.

Table 7 summarizes the applied machine learning algorithms, their features, used dataset for either malware detection or classification and accuracy levels. The number of families and samples used as a dataset varies a lot. For instance, the accuracy is higher when the set of 10 families or 3 families is experimented toward classification of a malware, see for instance [12] and [36] using 3 and 10 families and reaching at 80% and 97% accuracy, respectively. Similar reasoning holds for [18] and [19], tpr is above 0.9 when using 4 or 5 malware families. However, the study in [39] using the set of 32 families assures 66.8% in accuracy. The level in accuracy in [20] is very high but this methodology is NP-hard problem and requires significant resource usage. Our approach uses a fairly large set of families, samples, and features, it gives accurate and realistic results and it can be performed on an ordinary server (for example 8GB RAM, 4 CPU i5). We experimented runtime of 1.3 seconds to train and test of 101 new malware samples.

4. Conclusion

Currently, most malware samples are derived from existing ones and when these samples are armed with obfuscation techniques common security solutions can be easily evaded. This paper addresses the challenge of classifying malware samples by using runtime artifacts while being robust to obfuscation. Furthermore, the presented classification system is usable on a large scale in the real world due to its online machine learning methodology and its distributed architecture.

The proposed method uses run-time behaviors of an executable to build feature vector. We applied and evaluated five online machine learning algorithms with 17,900 samples belonging to 51 families. CW algorithm gives the most accurate results when compared to others. Its training and testing accuracy is reached at 94% and 92.5%, respectively.

This study is focused in particular on a scalable implementation for malware classification. The results of this research illustrate that runtime behavior modeling is a beneficial method for classifying malware. A useful scheme is presented by delivering efficient and accurate solutions to anti-virus companies or malware research institutes.

References

[1] Ashu S., Sahay S. K.. Evolution and detection of polymorphic and metamorphic malwares: a survey. Int J Comput Appl. 90(2).. 10.5120/15544-4098.

[2] Internet security threat report. 2016. Available at: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf, 2014, [accessed 17.01.2016].

[3] Sukwong O, Kim H, Hoe J. Commercial antivirus software effectiveness: an empirical study. Computer (Long Beach Calif) 2011;44(3):63–70.

[4] Tirli H, Pektas A, Falcone Y, Erdogan N. Virmon: a virtualization-based automated, dynamic malware analysis system. In: Proceedings of International Conference on Information Security and Cryptology (ISCTurkey); 2013. p. 57–62.

[5] Cuckoo foundation. Cuckoo SandboxAvailable at: http://www.cuckoosandbox.org/, [accessed 01.03.2017].

[6] Pektas A, Acarman T. A dynamic malware analyzer against virtual machine aware malicious software. Secur Commun Netw 2014;7(12). 22452257.

[7] RegShot. Registry compare utility. Available at: http://sourceforge.net/projects/regshot/, [accessed 01.03.2017].

[8] Pektas A, Eris M, Acarman T. Proposal of n-gram based algorithm for malware classification. In: Proceedings of the fifth International Conference on Emerging Security Information, Systems and Technologies; 2011. p. 7–13.

[9] Siddiqui M, Wang MC, Lee J. Detecting internet worms using data mining techniques. J Syst Cybern Inf 2009;6(6):48–53.

[10] Alazab M, Venkatraman S, Watters P, Alazab M. Zero-day malware detection based on supervised learning algorithms of API call signatures. In: Proceedings of the Ninth Australasian Data Mining Conference(AusDM); 2011. p. 171–82.

[11] Ding Y, Dai W, Yan S, Zhang Y. Control flow-based opcode behavior analysis for malware detection. J Comput Secur 2014;44:65–74.

[12] Mohaisen A, West AG, Mankin A, Alrawi O. Chatter: classifying malware families using system event ordering. In: Proceedings of the Communications and Network Security (CNS); 2014. p. 283–91.

[13] Salehi Z, Ghiasi M, Sami A. A miner for malware detection based on API function calls and their arguments. In: International Symposium on Artificial Intelligence and Signal Processing (AISP); 2012. p. 563–8.

[14] Chandramohan M, Tan HBK, Briand LC, Shar L, Padmanabhun BM. A scalable approach for malware detection through bounded feature space behavior modeling. In: Proceedings of Automated Software Engineering (ASE); 2013. p. 312–22.

[15] Bailey M, Oberheide J, Andersen J, Mao MZ, Jahanian F, Nazario J. Automated classification and analysis of internet malware. Recent Adv Intrusion Detect 2007:178–97.

[16] Rieck K, Holz T, Willems C, Düssel P, Laskov P. Learning and classification of malware behavior. Detect Intrusions Malware Vulnerability Assess. 2008:108–25.

[17] Zhao H, Xu M, Zheng N, Yao J, Ho Q. Malicious executables classification based on behavioral factor analysis. In: International Conference on e-Education, e-Business, e-Management, and e-Learning, IC4E'10; 2010. p. 502–6.

[18] Pirscoveanu RS, Hansen SS, Larsen TM, Stevanovic M, Pedersen JM, Czech A. Analysis of malware behavior: type classification using machine learning. In: 2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA); 2015. p. 1–7.

[19] Hansen SS, Larsen TM, Stevanovic M, Pedersen JM. An approach for detection and family classification of malware based on behavioral analysis. In: International Conference on Computing, Networking and Communications (ICNC); 2016. p. 1–5.

[20] Ki Y, Kim E, Kim HK. A novel approach to detect malware based on API call sequence analysis. Int J Distrib Sens Netw 2015:1–9.

[21] Hunt G, Brubacher D. Detours: binary interception of WIN 32 functions. In 3rd Usenix Windows NT Symposium; 1999.

[22] Pektas A, Acarman T, Falcone Y, Fernandez JC. Runtime-behavior based malware classification using online machine learning. In: In 2015 World Congress on Internet Security (WorldCIS); 2016. p. 166–71.

[23] Virustotal, online multiple AV scan service. Available at: http://www.virustotal.com/, [accessed 01.09.2016].

[24] GlusterFS. Available at: http://www.gluster.org/, [accessed 15.02.2016].

[25] Jubatus : Distributed online machine learning framework. Available at: http://jubat.us/en/, [accessed 09.10.2016].

[26] Sophisticated indicators for the modern threat landscape: an introduction to openIOC. Available at: http://openioc.org/resources/An_Introduction_to_OpenIOC.pdf, 2014, [accessed 17.12.2016].

[27] Kirillov I., Chase P., Beck D., Martin R.. Malware attribute enumeration and characterization. MITRE Corporation, Available at:https://maec.mitre.org/about/docs/Introduction_to_MAEC_white_paper.pdf, 2011, [accessed 19.10.2016].

[28] Virusshare, malware sharing platform. Available at: http://www.virusshare.com/, [accessed 01.09.2016].

[29] Crammer K, Dekel O, Keshet J, Shalev-Shwartz S, Singer Y. Online passive-aggressive algorithms. J Mach Learn Res 2006;7:551–85.

[30] Dredze M, Crammer K, Pereira F. Confidence-weighted linear classification. In: Proceedings of the 25th International Conference on Machine Learning (ICML); 2008. p. 264–71.

[31] Crammer K, Kulesza A, Dredze M. Adaptive regularization of weight vectors. Adv Neural Inf Process Syst 2009:414–22.

[32] Crammer K, Lee DD. Learning via gaussian herding. Neural Inf Process Syst 2010:451–9.

[33] Google corp., antivirus scan results for 8052d3912301ce691093cee448427547 in virustotal. Available at: https://www.virustotal.com/tr/file/cb1866afa08eb2a3f2cbd04068fbed2059c9e621b332a7376a16948d876b0c6c/analysis/, [accessed 30.09.2016].

[34] Available at: http://research.pektas.in, [accessed 15.02.2017].

[35] Landage J, Wankhade MP. Malware detection with different voting schemes, COMPUSOFT. Int J Adv Comput Technol 2014;3(1).

[36] Moonsamy V, Tian R, Batten L. Feature reduction to speed up malware classification. Inf Secur Technol Appl 2012;7161:176–88.

[37] Islam MR, Tian T, Batten L, Versteeg S. Classification of malware based on string and function feature selection. In: Proceedings of Cybercrime and Trustworthy Computing Workshop (CTC); 2010. p. 9–17.

[38] Zhong Y, Yamaki H, Takakura H. A malware classification method based on similarity of function structure. In: Proceedings of Applications and the Internet (SAINT); 2012. p. 256–61.

[39] Canzanese R, Kam M, Mancoridis S. Toward an automatic, online behavioral malware classification system. In: Proceedings of Self-Adaptive and Self-Organizing Systems (SASO); 2013. p. 111–20.

[40] Lin M, Chiu C, Lee Y, Pao H. Malicious URL filtering - a big data application. In: Proceedings of the IEEE International Conference on Big Data; 2013. p. 589–596.