Chapter 11

A BEHAVIOR-BASED APPROACH FOR MALWARE DETECTION

Rayan Mosli, Rui Li, Bo Yuan and Yin Pan

Abstract Malware is the fastest growing threat to information technology systems. Although a single absolute solution for defeating malware is improbable, a stacked arsenal against malicious software enhances the ability to maintain security and privacy. This research attempts to reinforce the anti-malware arsenal by studying a behavioral activity common to software – the use of handles. The characteristics of handle usage by benign and malicious software are extracted and exploited in an effort to distinguish between the two classes. An automated malware detection mechanism is presented that utilizes memory forensics, information retrieval and machine learning techniques. Experimentation with a malware dataset yields a malware detection rate of 91.4% with precision and recall of 89.8% and 91.1%, respectively.

Keywords: Malware, memory forensics, machine learning, handles

1. Introduction

The threat of malware is growing. The proliferation of electronic devices and the ever-increasing dependence on information technology have led to malware becoming an attractive tool for conducting criminal activities. Kaspersky Lab [10] reports that almost 250 million new and unique malware instances were detected during the second quarter of 2016 alone. Although substantial, the report was only able to present the amount of malware detected by anti-viral tools; it was not possible to estimate the total number of malware instances in the wild.

Current malware detection approaches focus on extracting unique signatures from captured malware samples and using the signatures in subsequent sightings of the same malware. This detection strategy is fast and has low false positive rates, but it is easily defeated by modifying

© IFIP International Federation for Information Processing 2017

Published by Springer International Publishing AG 2017. All Rights Reserved
G. Peterson and S. Shenoi (Eds.): Advances in Digital Forensics XIII, IFIP AICT 511, pp. 187–201, 2017.
DOI: 10.1007/978-3-319-67208-3_11

the malware code via encryption or packing [15]. Another strategy is to use a machine learning model to detect malware based on static malware features [24]. Although this malware detection strategy is more robust than signature-based detection, it can still be defeated [15].

The path to improving the detection of unknown malware started with the shift from signature-based detection to behavior-based detection. Behavior-based detection, which focuses on the activities of malware when it infects a system, can be implemented in two ways. The first involves extracting behavioral traits from the malware code statically; these traits are called malware semantics [6]. The second approach involves running malware in a sandbox environment and dynamically monitoring its behavior. System calls constitute an example of malware behavior that can be monitored dynamically and subsequently leveraged in malware detection [17, 21, 26]. Other behavioral features include file activity [22], registry activity [1] and API calls [27]. Behavior-based detection certainly improves the detection of unknown malware, but it is often slow and resource intensive because it requires running the malware in a sandbox. Furthermore, false positives are often a concern due to the misclassification of benign software that exhibits behavior similar to malware.

Several researchers have applied memory forensics to capture artifacts of malicious behavior that reside in memory [9, 23, 35, 36]. Memory forensics involves the analysis of a memory dump to extract evidence of malicious activity. An investigation using memory forensics has two stages: (i) memory acquisition; and (ii) memory analysis. In the memory acquisition stage, a digital forensic professional obtains a memory image via an acquisition tool such as Memoryze or Winpmem. The memory analysis stage attempts to find evidence of malicious activity using a tool such as Volatility or Rekall. This research employs memory forensics to extract information from memory images that is subsequently analyzed to detect malware.

The malware detection approach discussed in this chapter focuses on handles, abstraction pointers that are used to identify and access system objects without knowing their exact locations in memory. A resource such as a file, registry key or mutant requires a handle to be opened before it can be accessed. The handle must be closed when the resource is no longer required. Failing to release a handle may cause a handle leak, which can result in reaching the upper limit on the number of handles permitted by an application [29]. More than 30 resource object types can be identified using handles. Whenever a process requires such a resource, it must open a handle to the resource. The proposed approach uses the number of handles opened by a process to determine if it is potentially

malicious. After dynamically obtaining the handle data by running the software in a sandbox, machine learning is used to discriminate between benign and malicious uses of handles and to generalize handle usage behavior to detect previously-unknown malware.

2. Related Work

Research in malware detection can be categorized according to how malware analysis is conducted. Static analysis involves dissecting malware code and analyzing the instructions, imported libraries, metadata, and program functionality and structure. However, challenges arise if the malware is packed or encrypted. Nonetheless, this type of analysis offers the advantage of observing all the execution flows of the code regardless of the environment. Dynamic analysis, on the other hand, involves running the malware in a sandbox and monitoring its behavior. Although this analysis is not affected by encryption or packing, the malware behavior might differ according to the runtime environment.

2.1 Static Analysis

Santos et al. [32] used opcode sequences to train a support vector machine (SVM) classifier with a normalized polynomial kernel; features were extracted using term frequency to count the occurrences of opcodes in malware code. Saxe and Berlin [33] used byte entropy, portable executable (PE) imports and metadata to train deep neural networks to detect malware with dropout to prevent overfitting. Markel and Bilzor [20] also used metadata as features; they trained and compared different classifiers and found that a decision tree classifier outperformed naive Bayes and logistic regression classifiers on the particular data and feature sets. Nath and Mehtre [24] studied the performance of machine learning classifiers trained on static features; they concluded that using static features in malware detection faces several challenges such as encryption and packing, k-ary code and multistage loaders.

2.2 Dynamic Analysis

Pirscoveanu et al. [27] used the Cuckoo automated malware analysis tool to execute and monitor malware. They trained a random forest classifier using behavioral features (DNS requests, accessed files, mutexes, registry keys and API calls) and used INetSim to simulate an Internet connection for malware. Berlin et al. [3] used an n-gram bag of words with a sliding window to extract malware behavioral features from Windows audit logs and trained a logistic regression classifier on data generated by running and monitoring malware samples using Cuckoo.

Mohaisen et al. [22] developed AMAL, a malware detection and classification system. AMAL comprises two subsystems: (i) AutoMal, which runs malware samples and extracts features related to memory, filesystem, registry and network activity; and (ii) MaLabel, which vectorizes features and trains the classifiers. Park et al. [26] derived behavioral graphs from malware samples by running them in a sandbox and monitoring their system calls using Ether. They then created a graph for each malware family by observing a common sub-graph for malware instances belonging to the same family. In the detection phase, a matching process is used to determine the maliciousness of a file and the malware family to which it belongs (if the file is found to be malicious).

In a previous study, the authors of this chapter [23] examined registry activity, imported DLLs and called APIs to determine their potential as features for discriminating between benignware and malware. The most distinguishing features were determined, following which, machine learning models that utilize the features were trained to classify activity (processes) as benign or malicious. A detection rate of 96% was achieved by training a support vector machine classifier through the optimization of a hinge loss function. The support vector machine classifier was trained on registry activity data generated by software from both classes.

The use of handles in malware detection is relatively uncommon. Galal et al. [11] used handles to categorize different API calls according to their actions; the APIs either created handles, passed handles as arguments, released or closed handles or were handle-independent. Naval et al. [25], however, explicitly ignored handles along with all system call parameters. Park et al. [26] used handles to express dependencies between different kernel objects and their attributes. This chapter discusses the potential of handles to provide insightful views of program behavior based on the resources that are used. These insights are used to train a model to distinguish between benign software and malicious software based on the number of handles used for each resource.

3. Windows Handles and Objects

A handle is a pointer or reference to a Windows object [16]. Objects are managed by the Windows object manager, which is in charge of creating, deleting, protecting and tracking objects [31]. Every EPROCESS structure in memory contains a pointer in its *ObjectTable* member that points to a handle table, which contains pointers to all open objects used by the owning process. Each table has a *TableCode* that specifies the base address of the table and the number of levels in the table. A handle

table may contain up to three levels that, in theory, can carry up to 2^{29} handles. When more than one level exists in a handle table, only the last level points to objects. Otherwise, each entry in the preceding levels points to other tables.

A table also contains a member that holds the number of handles in the table. When a process calls an API such as `CreateFile`, a pointer to the created file is added to the process handle table and the index of the entry is returned. This index is the handle to the file, which is used by the process whenever the file is accessed. The *HandleCount* member of the handle table is incremented whenever a handle is added. Each entry in the handle table contains a pointer to the object header of the referenced object and a bit mask that expresses the access rights provided to the owning process. A subset of objects allow handles to be inherited by child processes from parent processes; an inherited handle has a unique value, but it points to the same object as the parent handle.

More than 30 object types are referenced by handles; observing the number of handles to each object type provides valuable insights into the resources that are used. The handles used by a process can be enumerated in several ways. One way is to do this programmatically by calling `NtQuerySystemInformation` with `SystemHandleInformation`. Alternatively, the Sysinternals *Handle* command line tool displays handle information about all processes, or about a single process if a process id (PID) is specified by the user [30]. Another approach is to use the Application Verifier tool from Microsoft to track process handle activity from start to finish. The proposed approach uses the Volatility `handles` plugin to extract handle information. This approach walks the handle table for a given process and displays its content. The `handles` plugin provides several options to filter the results: process id, EPROCESS structure offset, object type and object name. The process id was used to obtain the necessary data for processes known to be benign or malicious. Figure 1 shows a portion of a Volatility `handles` plugin output.

4. Malware Detection Using Handles

This section discusses malware detection using handles. The steps include collecting data, extracting features and training the machine learning models.

4.1 Experimental Setup

The dataset comprised 3,130 malware samples from the VirusShare malware repository [28]. Additionally, 1,157 benign software samples

```
Offset (V)   Pid Handle    Access Type       Details
---------- ----- ------ --------- ---------- -------
0x891cfea8  3104    0x4       0x3 Directory  KnownDlls
0x8439ff80  3104    0x8  0x100020 File       \Device\HarddiskVolume2\Users\
                                               victim\AppData\Local\Temp
0x84335368  3104    0xc  0x100020 File       \Device\HarddiskVolume2\Windows\
                                               winsxs\x86_microsoft.windows
0x8917fbd8  3104   0x10   0x20019 Key        MACHINE\SYSTEM\CONTROLSET001\
                                               CONTROL\NLS\SORTING\VERSIONS
0x85e0b778  3104   0x14  0x1f0001 ALPC Port
0x95cf1db8  3104   0x18       0x1 Key        \MACHINE|SYSTEM|CONTROLSET001\
                                               CONTROL\SESSION MANAGER
```

Figure 1. Output of the Volatility `handles` plugin.

were collected from various locations such as the Windows System32 directory and from software websites such as FileHippo.

The environment used for the analysis comprised a Ubuntu virtual machine that hosted four Windows 7 SP1 virtual machines using VirtualBox. The Ubuntu virtual machine was hosted on a Windows 10 machine using VMWare. Each Windows machine was set to have 1 GB RAM and one core.

Cuckoo [8] was used to automate the analysis process on the Ubuntu machine. The four Windows virtual machines were run concurrently, each with an instance of benign or malicious software. During the analysis task, a memory dump was produced of each Windows virtual machine along with a report with content and behavioral information about the sample. Furthermore, VirusTotal was used to scan each sample to ensure that the sample was labeled correctly as benign or malicious, and then determine the malware family to which it belonged. The majority of the samples were Trojans, but worms, viruses, backdoors and adware were also encountered. A portion of the dataset was classified as being malware, but no consensus was reached by VirusTotal about the families to which all the samples belonged. These samples were included in the malware dataset, but were labeled as unclassified instead of as a malware family.

INetSim [14] was used to simulate an Internet connection to increase the chances of the malware behaving correctly. However, due to their anti-virtual-machine functionality, 668 malware samples terminated instantly after being launched; this left 2,462 malware samples to be used in the experiments. Although discarding malware with anti-virtual-machine functionality from the dataset omits such behavior from the classifiers, the increasing popularity of virtualization in the information technology sector is making malware with anti-virtual-machine functionality less common [18].

Handle data was extracted from the memory dumps of machines with benignware or malware using Volatility. Every EPROCESS structure in memory contains a pointer to a handle table specific to the owning process. Volatility outputs the handle information by walking the handle table of a specified process or of all processes if no process id was specified when running the `handles` plugin [16]. The process ids used to filter the results were provided by Cuckoo; the main process id in addition to the process ids of spawned processes were included in the Cuckoo report. The Volatility `handles` plugin outputs a table with six columns: (i) virtual offset of the handle in memory; (ii) process id of the owning process; (iii) handle offset in the process handle table; (iv) access granted to each object with a handle; (v) type of object pointed to by the handle; and (vi) details about the object, if available. All the Volatility results were stored in text files, where each text file contained information about the handles used by a single process.

4.2 Vectorizing the Handle Data

The term frequency-inverse document frequency (TF-IDF) [19] was used to extract measurable features from the Volatility `handles` output; the extraction and model training was implemented using scikit-learn [5]. A vocabulary was created comprising the handle types to be extracted from the handle text files. A list of all the possible terms in the vocabulary was obtained from Schuster [34]. Subsequently, the term frequency-inverse document frequency, which counts the occurrence of each vocabulary term in each text file, and then weights the importance of the term according to the number of times the term occurs across all the documents, was computed for all the handles data. This yielded a $3,619 \times 31$ matrix, each row representing a sample and each column representing a term. A total of 58,652 non-zero entries were present in the matrix, making the matrix 52.28% dense. To avoid division by zero, the `smooth_idf` option was set to true; this option adds one document to the corpus with every term in the vocabulary appearing once. Zero entries appearing in the matrix were largely the result of ten terms that did not appear in any document. These terms were discarded before training the models, resulting in a $3,619 \times 21$ matrix with a density of 77.17%.

4.3 Model Training

For evaluation purposes, the dataset was divided into two subsets, one for training and one for testing. A total of 724 samples were used for testing (20% of the dataset) and 2,895 samples were used for training.

A stratified split was used to generate the test set, which resulted in a balanced representation of both classes.

Three machine learning models were compared: (i) k-nearest neighbor (KNN) [2]; (ii) support vector machine (SVM) [7]; and (iii) random forest [12]. The k-nearest neighbor approach classifies each data point according to its neighbors; a number of options must be considered when training this classifier, including the number of neighbors to be evaluated and the method for assigning weights to the neighbors. The support vector machine is a discriminative model that searches for a hyperplane with maximum separation between the data points from different classes; the hyperplane is then used to classify new data points according to the side of the hyperplane where they fall. Random forest is an ensemble approach that trains multiple decision trees and outputs a decision according to the predictions of all the trees.

Accuracy, precision and recall were used as evaluation metrics for the three machine learning models. The exhaustive grid search approach was employed to determine the parameter values that produced the highest detection rates for the models. To perform the exhaustive grid search, a parameter space was created for each model that was populated with the values to be searched. Table 1 lists the parameter values tested for each machine learning model.

The k-nearest neighbor approach achieved the highest accuracy using three neighbors, the ball tree algorithm to find neighbors and distances as weights. In the case of the SVM, a radial basis function (RBF) kernel gave the highest accuracy; the numbers of support vectors used were 527 for the benign class and 625 for the malicious class. The random forest approach performed best with 25 decision trees. After determining the best parameter values for each model, the precision and recall were calculated to measure the model performance with regard to false positives and false negatives. Table 2 summarizes the performance of the three machine learning models.

Table 3 shows the confusion matrix for the random forest classifier with the predicted and true labels.

5. Results and Analysis

Observations of the use of handles by benign and malicious software can reveal their potential for helping discriminate between the two classes. For example, section handles are used differently by benignware and malware. A section is a region of memory that can be shared by multiple processes. It is used by the Windows loader when loading a module into process address space [4]. A section is also used for inter-

Table 1. Exhaustive grid search parameter space.

Model	**Parameter**	**Value**
KNN	Algorithm for finding neighbors	Ball tree KDtree Brute force
	Number of neighbors	3 4 5 6 7
	Weights of neighbors	Uniform Distance
SVM	Penalty term	1 0.75 0.50 0.25
	Kernel type	Linear Polynomial RBF Sigmoid
	Degree of polynomial	1 2
Random Forest	Max feature split algorithm	Auto Square root Logarithmic None
	Number of decision trees	5 10 15 20 25

Table 2. Performance of the KNN, SVM and random forest models.

Learning Model	**Accuracy**	**Precision**	**Recall**
KNN	0.910	0.892	0.899
SVM	0.911	0.899	0.920
Random Forest	0.914	0.898	0.911

Table 3. Confusion matrix for the random forest classifier.

		Predicted	
		0	**1**
True	**0**	29.5%	3.1%
	1	5.3%	61.8%

process communication (IPC), where a memory-mapped file is shared by two or more processes. When used with malicious intent, sections provide a means for injecting code into the address spaces of other processes. The different usage of sections by benign and malicious software explains the difference in the numbers of handles used by the two types of software. In the experiments, the average number of section objects used by benignware was 8.48 whereas the average number for malware was 27.12. Therefore, when training the random forest classifier, section features were at the top of the decision trees; this affected the largest fraction of sample predictions.

The number of process handles used by software is another indicator of maliciousness. A process handle is often obtained when a new process is created using the `CreateProcess` function. Furthermore, a handle to a process can also be retrieved by passing a process id to the `OpenProcess` function. Malware uses process handles to gain access to other victim processes with the goal of injecting, hollowing, terminating or hooking [13]. In the experiments, the average number of process handles used by legitimate software was 0.81 whereas the average number used by malware was 2.43. Consequently, process handles became the second most prominent term when training the random forest classifier.

Mutants are objects that can also help distinguish between benignware and malware. Mutants are used for mutual exclusions; specifically, to control access to shared system resources. A mutant handle is acquired by calling the function `OpenMutex` and is released by calling the function `ReleaseMutex`. Mutants are often used by legitimate software to avoid conflicts between multiple threads. However, malware samples use them to prevent the re-infection of already-infected resources, which could result in undesirable results. In the experiments, the average number of mutant objects used by benignware was 11.35 whereas the average number used by malware was 21.84. Table 4 shows the use of handles by benignware and malware.

To determine the effects of an imbalanced dataset (benignware: 1,157 and malware: 2,462) on the machine learning model, the experiments

Table 4. Use of handles by benignware and malware.

Object Type	Benignware		Malware	
	Average	Variance	Average	Variance
Desktop	1.92	3.73	1.85	0.54
Device	22.8	710.38	25.34	148.42
Directory	2.21	0.25	2.35	0.25
Event	70.58	7468.62	94.92	2516.89
File	22.80	710.44	25.34	148.42
IOCompletion	1.06	1.36	2.31	0.93
Job	0.01	0.01	0.38	0.23
Key	32.79	873.35	49.66	277.51
KeyedEvent	0.56	0.26	0.93	0.06
Mutant	11.35	227.5	21.84	97.71
Port	7.51	57.27	12.76	25.36
Process	0.81	61.78	2.43	16.71
Section	8.48	70.13	27.12	156.99
Semaphore	10.91	173.83	12.59	43.01
Thread	14.38	668.91	20.61	98.75
Timer	2.01	4.30	2.85	0.75
Token	0.45	6.63	0.09	0.94
WindowStation	2.01	0.01	2.01	0.01
WmiGuid	0.18	0.14	0.02	0.02

were performed multiple times with a balanced dataset. This was accomplished by randomly discarding malware samples to reduce the number to 1,157. During each run, a different subset of malware samples was discarded. The performance of the classifiers trained on the balanced datasets was only slightly lower than the classifiers trained on the original dataset. This leads to the conclusion that significant behavior from the malware dataset can be captured using a smaller dataset.

6. Conclusions

This research has demonstrated that handles, which capture the behavioral activity of software, can be used to detect malware. Specifically, malware uses resources differently from benignware and this fact can be used to train classifiers to categorize processes as malicious or benign. In the experiments, Cuckoo was used to automate the execution and monitoring of malware and benignware and to dump memory images. Volatility was used to extract the handle information from the memory dumps, which was then analyzed to determine the different uses of handles by the two classes of software. Three machine learning models,

k-nearest neighbor, support vector machine and random forest, were used to train the classifiers. Random forest outperformed the k-nearest neighbor and support vector machine models with a detection rate of 91.4%, precision of 89.8% and recall of 91.1%.

The experimental results demonstrate the efficacy of using handles to detect malware. However, the approach is reactive in that it is applied after the system has been infected. Nevertheless, one use case for the approach is as a second layer of defense if signature-based detection fails. The second use case is in forensic investigations, where malware detection and analysis are routinely performed. In fact, the approach can be applied to alleviate the cumbersome task of detecting malware in a large number of seized machines.

This research has focused on the types of objects referenced in handle tables. Information provided in the access rights and details columns of the Volatility `handles` plugin output was not considered. The details column provides in-depth information about objects, such as the registry key accessed by the process and the file path to which a handle is opened. File objects, in particular, may not be actual files – they may be devices treated as files due to similar read and write operations. Such granular details could significantly improve the performance of the classifiers. This exploration is a topic of future research.

Another topic for future research is the identification of other behavioral artifacts that may be used to distinguish malware from benignware. Zaki and Humphrey [37], who studied kernel-level artifacts left by rootkits, discovered that callbacks are more suspicious than other artifacts such as SSDT hooks. Future research will investigate the use of callbacks and other artifacts in developing classifiers with improved malware detection rates, precision and recall.

References

[1] M. Aghaeikheirabady, S. Farshchi and H. Shirazi, A new approach to malware detection by comparative analysis of data structures in a memory image, *Proceedings of the First International Congress on Technology, Communication and Knowledge*, 2014.

[2] N. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, *The American Statistician*, vol. 46(3), pp. 175–185, 1992.

[3] K. Berlin, D. Slater and J. Saxe, Malicious behavior detection using Windows audit logs, *Proceedings of the Eighth ACM Workshop on Artificial Intelligence and Security*, pp. 35–44, 2015.

[4] B. Blunden, *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*, Jones and Bartlett Learning, Burlington, Massachusetts, 2013.

[5] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt and G. Varoquaux, API design for machine learning software: Experiences from the scikit-learn Project, *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[6] M. Christodorescu, S. Jha, S. Seshia, D. Song and R. Bryant, Semantics-aware malware detection, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 32–46, 2005.

[7] C. Cortes and V. Vapnik, Support-vector networks, *Machine Learning*, vol. 20(3), pp. 273–297, 1995.

[8] Cuckoo Foundation, Cuckoo Sandbox (`www.cuckoosandbox.org`), 2016.

[9] B. Dolan-Gavitt, A. Srivastava, P. Traynor and J. Giffin, Robust signatures for kernel data structures, *Proceedings of the Sixteenth ACM Conference on Computer and Communications Security*, pp. 566–577, 2009.

[10] D. Emm, R. Unuchek, M. Garnaeva, A. Ivanov, D. Makrushin and F. Sinitsyn, IT Threat Evolution in Q2 2016, Kaspersky Lab, Moscow, Russia, 2016.

[11] H. Galal, Y. Mahdy and M. Atiea, Behavior-based features model for malware detection, *Journal of Computer Virology and Hacking Techniques*, vol. 12(2), pp. 59–67, 2016.

[12] T. Ho, The random subspace method for constructing decision forests, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20(8), pp. 832–844, 1998.

[13] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*, Pearson Education, Upper Saddle River, New Jersey, 2006.

[14] T. Hungenberg and M. Eckert, INetSim: Internet Services Simulation Suite (`www.inetsim.org`), 2007.

[15] B. Klein and R. Peters, Defeating machine learning – What your security vendor is not telling you, presented at *Black Hat USA*, 2015.

[16] M. Ligh, A. Case, J. Levy and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux and Mac Memory*, John Wiley and Sons, Indianapolis, Indiana, 2014.

[17] Y. Lin, Y. Lai, C. Lu, P. Hsu and C. Lee, Three-phase behavior-based detection and classification of known and unknown malware, *Security and Communication Networks*, vol. 8(11), pp. 2004–2015, 2015.

[18] J. Luttgens, M. Pepe and K. Mandia, *Incident Response and Computer Forensics*, McGraw Hill Education, New York, 2014.

[19] C. Manning, P. Raghavan and H. Schutze, *An Introduction to Information Retrieval*, Cambridge University Press, Cambridge, United Kingdom, 2008.

[20] Z. Markel and M. Bilzor, Building a machine learning classifier for malware detection, *Proceedings of the Second Workshop on Anti-Malware Testing Research*, 2014.

[21] M. Masud, S. Sahib, M. Abdollah, S. Selamat and R. Yusof, Analysis of features selection and machine learning classifier in Android malware detection, *Proceedings of the International Conference on Information Science and Applications*, 2014.

[22] A. Mohaisen, O. Alrawi and M. Mohaisen, AMAL: High-fidelity, behavior-based automated malware analysis and classification, *Computers and Security*, vol. 52, pp. 251–266, 2015.

[23] R. Mosli, R. Li, B. Yuan and Y. Pan, Automated malware detection using artifacts in forensic memory images, *Proceedings of the IEEE Symposium on Technologies for Homeland Security*, 2016.

[24] H. Nath and B. Mehtre, Static malware analysis using machine learning methods, *Proceedings of the Second International Conference on Recent Trends in Computer Networks and Distributed Systems Security*, pp. 440–450, 2014.

[25] S. Naval, V. Laxmi, M. Rajarajan, M. Gaur and M. Conti, Employing program semantics for malware detection, *IEEE Transactions on Information Forensics and Security*, vol. 10(12), pp. 2591–2604, 2015.

[26] Y. Park, D. Reeves and M. Stamp, Deriving common malware behavior through graph clustering, *Computers and Security*, vol. 39(B), pp. 419–430, 2013.

[27] R. Pirscoveanu, S. Hansen, T. Larsen, M. Stevanovic, J. Pedersen and A. Czech, Analysis of malware behavior: Type classification using machine learning, *Proceedings of the International Conference on Cyber Situational Awareness, Data Analytics and Assessment*, 2015.

[28] J. Roberts, VirusShare Project (`virusshare.com`), 2017.

[29] M. Russinovich, Pushing the limits of Windows: Handles, *Mark's Blog* (`blogs.technet.microsoft.com/markrussinovich/2009/09/29/pushing-the-limits-of-windows-handles`), September 29, 2009.

[30] M. Russinovich, Sysinternals Suite, Microsoft TechNet, Redmond, Washington (`technet.microsoft.com/en-us/sysinternals/bb842062.aspx`), 2017.

[31] M. Russinovich, D. Solomon and A. Ionescu, *Windows Internals*, Microsoft Press, Redmond, Washington, 2012.

[32] I. Santos, F. Brezo, X. Ugarte-Pedrero and P. Bringas, Opcode sequences as representation of executables for data-mining-based unknown malware detection, *Information Sciences*, vol. 231, pp. 64–82, 2013.

[33] J. Saxe and K. Berlin, Deep neural network based malware detection using two dimensional binary program features, *Proceedings of the Tenth International Conference on Malicious and Unwanted Software*, pp. 11–20, 2015.

[34] A. Schuster, Enumerate Object Types, *Computer Forensic Blog* (`computer.forensikblog.de/en/2009/04/enumerate-object-types.html`), April 7, 2009.

[35] J. Stuttgen and M. Cohen, Anti-forensic resilient memory acquisition, *Digital Investigation*, vol. 10(S), pp. S105–S115, 2013.

[36] T. Teller and A. Hayon, Enhancing automated malware analysis machines with memory analysis, presented at *Black Hat USA*, 2014.

[37] A. Zaki and B. Humphrey, Unveiling the kernel: Rootkit discovery using selective automated kernel memory differencing, presented at the *Virus Bulletin Conference*, 2014.