2019 International Conference on Advancements in Computing (ICAC)
December 5-6, 2019. Malabe, Sri Lanka

An Automated Tool for Memory Forensics

Mifraz Murthaja, Benjamine Sahayanathan, A.N.T.S.Munasinghe, Diluxana Uthayakumar, Lakmal Rupasinghe, Amila Senarathne
Faculty of Computing, Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
mmtech95@gmail.com, sahayanathanbenjamine.sb@gmail.com, tharikasewwandi92@gmail.com, diluxanauthayakumar@gmail.com, lakmal.r@sliit.lk, amila.n@sliit.lk

***Abstract*—In the present, memory forensics has captured the world's attention. Currently, the volatility framework is used to extract artifacts from the memory dump, and the extracted artifacts are then used to investigate and to identify the malicious processes in the memory dump. The investigation process must be conducted manually, since the volatility framework provides only the artifacts that exist in the memory dump. In this paper, we investigate the four predominant domains of registry, DLL, API calls and network connections in memory forensics to implement the system 'Malfore,' which helps automate the entire process of memory forensics. We use the cuckoo sandbox to analyze malware samples and to obtain memory dumps and volatility frameworks to extract artifacts from the memory dump. The finalized dataset was evaluated using several machine learning algorithms, including RNN. The highest accuracy achieved was 98%, and it was reached using a recurrent neural network model, fitted to the data extracted from the DLL artifacts, and 92% accuracy was reached using a recurrent neural network model, fitted to data extracted from the network connection artifacts.**

Keywords*—Memory forensics, malware, cuckoo sandbox, volatility, machine learning, deep learning, feature selection

I. INTRODUCTION

Malware has become one of the most expeditious threats to all the information systems, and the number of malicious programs is growing daily, since the current world is transitioning rapidly into digitalization. According to the AV-TEST institute statistics [1], malware creation has increased from 47.05m (in 2010) to 856.62m (in 2018). Hence, the complexity of its identification is rising every moment. Currently, most of the anti-virus software use detection techniques such as signature-based detection, heuristic-based detection and behavior-based detection [2][3]. However, each of these techniques has its own drawbacks. For example, the signature-based technique can only be used for known malware. The heuristic-based technique uses a method which can compare the identified malware signatures with other signatures [3]. Sometimes, a legitimate process or behavior can be identified as malware only because it is behaving as a malware. Considering all these factors, we decided to use memory forensics to detect malware in an efficient way.

As an incident response, the operation of collecting evidence from a particular computing device and analyzing and reporting the digital data in a legally admissible way, is referred to as computer forensics. In particular, the copy of the main memory (RAM) is taken as a digital copy (referred to as memory dump) for granular investigation, for incidents such as malware attacks and virus infections. Such investigations are referred to as memory forensics. Simply, memory forensics is finding digital evidence to identify any malicious activities/behaviors in the memory dump. As an incident response, the forensic investigators extract the details from the memory dump and look for any abnormal behaviors of processes to identify malicious/suspicious processes. In the live analysis, process details such as terminated processes and cache details are ignored, and it is hard to track the behavior of the malicious process due to its packed nature and obfuscation techniques. While in memory forensics analysis, since the memory dump is a snapshot of a particular state of the main memory, the investigation process can be performed while preserving the state of the affected system [5]. Forensic investigators conduct malware identification manually, with the help of the volatility framework [4]. Input for the volatility framework is a memory dump, which can be taken through the operating system or by using some tools such as WinPMEM, DUMPIT and sandboxes.

In the context of the volatility framework, a system can read the contents inside the dump by converting the raw information into arrays, bitmaps, linked lists, doubly linked lists and the basic data structures, according to the RAM offset values. Python compilers convert the memory dump's raw information into python readable lists and array formats that helps the volatility to create Windows objects. The volatility framework marks the multiple Windows object structures that represent file, process, symbolic links, tokens, threads, mutant, Windows stations, desktops, drivers, keys and types with the object header, containing the information regarding the pointer count, handle count, index type, info mask, security descriptor and body. In the objects created by the volatility, _POOL_HEADER refers to the objects belonging to the kernel space or user space, and one object can link with multiple objects which belong to the kernel and user spaces, making it helpful for the malware analysis. Volatility consists of a different profile of hardware architectures of the OS. This is because metadata, system call information, constant values, native type languages and system maps are different when analyzing the object classes [7].

In this research, the four main feature types extracted using the volatility framework are registry activities, dynamic link libraries (DLLs), application programming interface (API) calls [2] and networking behaviors. These have been considered to analyze the memory dump to identify the malicious processes. The registry stores the required information to run applications. It is one of the core areas of the Windows OS that ensures the drivers are properly loaded in order to perform its tasks. It implies, for Windows to perform a particular task, the task modules must be loaded into the registry [8]. Similarly, for the malware to perform its intended tasks, it requires the registry to load the module.

The DLLs are minute sized units that represent the process. Windows APIs create the set of DLLs to execute the corresponding functions of the Windows program. The DLL information stored in the main memory is extremely helpful for identifying the benign and malware process and to cluster the malicious programs [9].

Windows API calls play a crucial role in process execution. As all the user applications run on user space, it requires system calls to request a service from the kernel space, such as read, write and delete, to perform sensitive tasks. The API calls help user space processes to interact with kernel space as well the other processes. These APIs aren't sufficient for identifying/detecting the malware but they can be helpful in identifying the behavior of the malware [10].

Furthermore, most of the malware uses networking services to communicate with the command and control centers, to spread malware to other machines and to create the backdoors. Therefore, analyzing the network packets available in the NetBIOS will help identify the malware for the researchers [7].

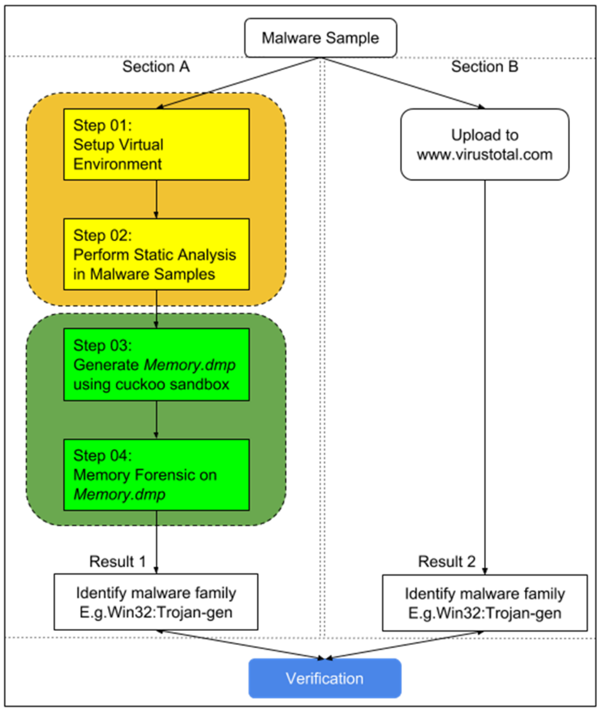
Presently, even though artificial intelligence is becoming influential, and advanced malware detection techniques are being implemented for live malware analysis [6], malware detection in memory forensics is not fully automated. Mosli et al. [2] proposed an approach to automate malware detection using volatility and Rekall [11], achieving an accuracy of 96%. In this paper, we propose a system 'Malfore,' which will make the entire process of memory forensics automated considering the registry, DLL, API call and network artifacts, using the volatility framework and increasing the accuracy of malware detection.

II. BACKGROUND

Nowadays, detection of malware has become much more complicated due to obfuscation techniques, such as metamorphism and the polymorphism [12]. Malware analysis can be divided into static and dynamic analyses, where the basis of the dynamic analysis is behavior analysis. Executing the binary code of malware in a controlled environment, such as a sandbox environment, can be classified under dynamic analysis [12]. Lim et al. proposed a unified framework to integrate static and dynamic analysis with machine learning techniques [12]. In memory forensics, static analysis accedes a conspicuous place rather than live analysis [13], and it is gaining more prominence since it provides a reliable and trusted platform for detecting malware [14]. Using memory forensics, artifacts can be extracted preserving the state of the digital shreds of evidence without any destruction.

However, malware finders are forced to analyze new malware manually, due to the inadequacy of automated memory forensics tools [2]. The Yara rules adorn a major place in identifying and classifying malware by creating rules that look for certain characteristics. Detecting malicious programs using Yara rules can be classified under static analysis, as it is similar to the signature-based method. Therefore, advanced malware behaviors still could not be identified due to obfuscation techniques. For advanced malware detection, especially for new malware, a novel approach was presented, based on the heuristic method.

In computer forensics, the source can either be disk image or RAM image. Most of the malware creators try to hide the footprints of malware in the disk, while the execution is transferred into the RAM [14]. However, the RAM image provides better results in identifying the malicious processes, as all the running processes must reside on the RAM to perform its intended task. Malware requires running in the system for execution, and it leaves footprints which can be noted as shreds of evidence and symptoms [14]. This evidence can be identified by analyzing suspicious process behaviors, recently accessed registries and DLLs as well as suspicious or hidden network connections and sockets. With the help of these indications, advanced malware can be identified effectively.

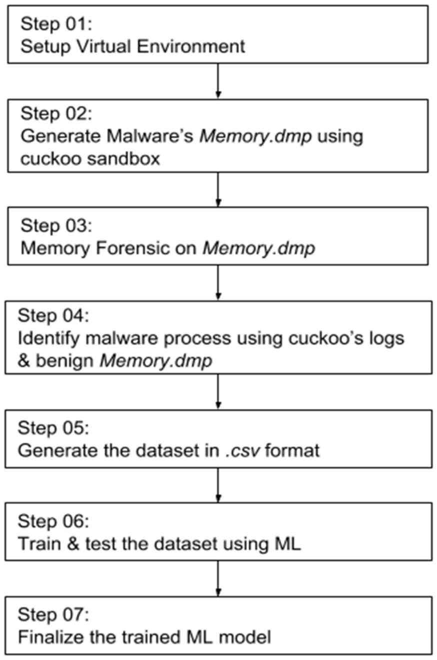


Malware Sample
Section A
Section B
Step 01: Setup Virtual Environment
Step 02: Perform Static Analysis in Malware Samples
Step 03: Generate *Memory.dmp* using cuckoo sandbox
Step 04: Memory Forensic on *Memory.dmp*
Upload to www.virustotal.com
Result 1
Result 2
Identify malware family E.g.Win32:Trojan-gen
Identify malware family E.g.Win32:Trojan-gen
Verification

Fig.1. Malware family identification framework [14]

Rathnayaka and Jamdagni proposed a hybrid malware analysis framework to analyze complicated malware. It analyzes the malware manually and verifies the results with the VirusTotal parallelly [14], as explained in Fig. 1. As a result of this analysis, they were able to achieve an accuracy of 90% [14].

III. METHODOLOGY



Step 01: Setup Virtual Environment
Step 02: Generate Malware's *Memory.dmp* using cuckoo sandbox
Step 03: Memory Forensic on *Memory.dmp*
Step 04: Identify malware process using cuckoo's logs & benign *Memory.dmp*
Step 05: Generate the dataset in *.csv* format
Step 06: Train & test the dataset using ML
Step 07: Finalize the trained ML model

Fig. 2. System implementation diagram

In order to collect malicious and benign memory dumps, we used 400 malware samples collected from VirusTotal, VirusShare and Kaggle, and 100 benign samples created by us, consisting of different types of files such as word documents, PDFs, audio/video files, images, etc. The collected memory dumps were then used to plot malicious information, based on registry activities, DLLs, APIs calls and network behaviors. Finally, the machine learning model was trained using the collected features to identify the benign and malicious processes. Eventually, if a user submits their memory dump, the Malfore system can predict the indication of any malicious behaviors of the processes existing in the memory dump. Fig. 2 describes the system implementation diagram of Malfore. The implementation of the system can be organized into four predominant sectors: data acquisition, feature extraction, feature selection and model training. The implemented system is then used to analyze the memory dump, as explained in Fig. 3.

A. Data Acquisition

Since malware should be analyzed in an isolated environment, we decided to use the sandbox environment. Cuckoo sandbox, DroidBox and Malwasm were used to analyze the malware, and Droidbox was used to evaluate the android applications. Malwasm is based on cuckoo, and it helps to log all the movements of malware and store that information in the database, which can be accessed via the web. Hence, when comparing to other sandboxes, the cuckoo sandbox was identified as the most suitable platform, as per the requirements of Malfore. The cuckoo sandbox was used in this research to acquire the memory dumps which are injected by malicious programs [15].

In this paper, the volatility framework is used to extract artifacts required for the DLL analysis, registry analysis, API analysis and network analysis. In order for the volatility framework to analyze DLLs and extract the details about the processes of the system and the loaded DLLs of the processes, it dumps all the DLLs loaded from all the processes including hidden ones to check the memory of csrss.exe and conhost.exe [18]. Furthermore, commands executed on cmd.exe or the backdoors used can be identified using the volatility framework. It helps gain the visibility of the activities performed by the attacker in the victim's machine. The data included in PE files, imported or exported functions, DLLs, logs, IE history and a list of kernel drivers can be extracted from the memory dumps using the volatility framework. For the registry analysis, all the keys and values can be extracted. Moreover, the volatility aids in locating the virtual addresses of the registry hives, provides the details about the accessed registries from the processes, extracts the decrypted cached domain credentials and dumps the LSA secrets and shellbag details. In the context of network analysis, artifacts of the activated TCP connections, terminated network connections and listening sockets in the TCP and UDP are analyzed. In order to perform the API analysis on the memory dump, hidden or injected codes/DLLs in the user mode memory, API hooks, open handles and thread details are acquired.

All the dumps analyzed in this research are from the Windows 7 64-bit version OS. The cuckoo sandbox agent was configured in Windows 7 to inject the malware samples and to take the memory dumps. The Windows virtual machine was configured as an end-user computer system, consisting of regular applications such as word processing, editing, different frameworks, torrent downloaders, etc. We obtained a list of regular applications using a Google form survey from 500 people. The Windows virtual machine was fully updated after the installation of all the regular applications. Finally, a snapshot of the fully configured Windows virtual machine was taken as the base state, to inject the malware and benign samples.

The fully configured sandbox was then injected with malware and benign samples, individually, to the base state, and 500 different memory dumps consisting of 400 malware, and 100 benign sample injected memory dumps were collected.

B. *Feature Extraction*

As mentioned in Section III (A), the cuckoo sandbox was used to analyze the malware samples on Windows 7. However, the report generated by it was not used in this research. Instead, we only used the cuckoo sandbox to inject the malicious and benign software, execute it in the Windows OS and to obtain the memory dump. There are two primary reasons for not using the report generated by cuckoo sandbox: (i) The scoring system in cuckoo is an alpha feature, and it requires further implementation. To prove this, an executable which only prints a "Hello World" statement is injected into cuckoo, and the cuckoo sandbox reported it as dangerous and the score was 7.8, whilst for some malware executables, the score was around 4. (ii) Since this research is on memory forensics, the entire artifacts must be extracted from the memory dump, and the same methodology should be implemented in the final system to predict the malicious processes in the given memory dump. Furthermore, obtaining the API call, DLL, registry and network connection artifacts separately, is required. Although, the cuckoo sandbox dataset contains the whole data extracted from memory as well as data extracted through some other ways, such as the tcpdump as a collection.

As mentioned in Section III (A), we used the volatility framework to extract artifacts from the memory dump. Each command or plugin available in the volatility provides artifacts in their own way. Hence, for us to use the artifacts of the memory dump, we created some scripts to extract the API call artifacts, DLL artifacts, registry artifacts and the network artifacts, using the existing plugins in the volatility and combined and processed the artifacts for each domain separately, such that our machine learning models could be used. Furthermore, we customized some plugins in the volatility, as required. The script for the API call artifacts extracted the IAT, EAT, inline style hooks in the user mode or kernel mode, CALLs and JMPs to direct and indirect locations, threads, executed commands, console information, etc. The script for the DLL artifacts extracted the loaded DLLs, open handles, security identifiers, etc. The script for the registry artifacts extracted the registry hives, registry keys, etc. Finally, the script for network artifacts extracted the IP addresses, port numbers, flags, etc. Afterwards, we processed the list of flag names and value pairs into flag values as the individual features.

In the final stage of feature extraction, we obtained 500 x 4 different datasets, consisting of the artifacts from 400 malicious and 100 benign memory dumps for the four domains: APIs, DLLs, registries and network connections. Following this, we obtained an additional dataset to identify the system processes and the startup processes, consisting of the artifacts extracted from the base memory dump without any injected processes. All the processes in the 100 datasets are benign, including the processes of the injected benign executables and the system processes. Even though the malicious datasets contain malicious processes, not all the processes are malicious, since it consists of processes of injected executables as well as the system processes. Therefore, only the processes of injected malicious executables should be labeled as malicious. Furthermore, the system processes for all the 500 datasets almost remain the same. Consequently, to reduce the redundancy, only the injected processes from all the 500 datasets were filtered and combined, along with the dataset extracted from the base memory dump, consisting all the system and startup processes. The cuckoo analysis log is used to filter the processes of the injected executables from the datasets, except for network connection datasets. We created a script to accomplish this task which read the analysis log file related to each dataset, obtained the PID of the injected executable, filtered the data for the identified PID and finally combined all the filtered data from 400 datasets into, labelling the tuples as benign/malicious.

Although, since the network-related artifacts are not based on the PID, the cuckoo analysis log cannot be used to filter the network artifacts of injected executables. Moreover, the network datasets contain the network connections used by cuckoo as well. Hence, to filter only the network connections used by the injected executable, we created some benign memory dumps by injecting plain text files, simple print statement executable, PDF files, etc. Since the same snapshot was used to analyze every injected executable, the offset of the system processes remains the same. Therefore, the combined malicious datasets were subtracted by the combined benign datasets to filter the malicious ones. Additionally, all the network connections in the benign datasets, including the connections used by cuckoo, were considered benign connections. At last, four different datasets were obtained for four domains, each consisting of the processes/network connections of the injected malicious and benign executables and the system and startup processes. The number of features and tuples obtained in the final datasets are as follows:

i.	API	: 182 features and 14000 tuples
ii.	DLL	: 141 features and 2800 tuples
iii.	Registry	: 39 features and 2500 tuples
iv.	Network	: 35 features and 1700 tuples

C. *Feature Selection*

To preprocess the data for machine learning, the null values were replaced with zeroes, and the categorical fields were converted to numerical fields using the MongoDB, to ensure that the converted numerical values remained the same, even in the final system, to avoid redundancy and to maintain consistency. Additionally, when the categorical field contains a numeric value, it remains the same in our customized algorithm. To select the golden features, finalized datasets were fed to the filter based feature selection algorithm (FBFS) [17]. For the FBFS algorithm, the Pearson correlation, mutual information, the Kendall correlation, the Spearman correlation, the chi-squared test, the Fisher score and counter-based metrics were used. The filtered feature sets were trained with the Gaussian naive Bayes (GNB), support vector clustering (SVC), K-nearest neighbor classifier (KNN), Logistic Regression (LR), Decision Tree Classifier (DTC), random forest classifier (RFC) and linear discriminant analysis (LDA), using the Scikit-Learn python machine learning library. The FBFS metric, which had maximum accuracy scores out of the average scores of all the algorithms, was selected as the best FBFS metric. Moreover, the models were evaluated using the K-fold cross-validation method with 10 tuples per split. Tables I describe the accuracy scores obtained for the different FBFS metrics trained with different machine learning algorithms for DLL artifacts. The similar tables are obtained for Registry, APIs and Network artifacts. Table II indicates the finalized feature sets for detecting malicious processes using API artifacts, DLL artifacts, registry artifacts and network connection artifacts.

TABLE I
SUMMARY OF ACCURACY SCORES OBTAINED FOR DIFFERENT FBFS FEATURE SETS TRAINED WITH DIFFERENT ALGORITHMS FOR DLL ARTIFACTS

	GNB	**SVC**	**KNN**	**LR**	**DTC**	**RFC**	**LDA**	**Average**
PC	0.93	0.83	0.92	1.00	1.00	1.00	0.97	0.95
MI	0.94	0.99	0.99	1.00	1.00	1.00	0.97	0.99
KC	0.97	0.80	0.92	1.00	1.00	1.00	0.97	0.95
SC	0.94	0.83	0.93	1.00	1.00	1.00	0.97	0.95
CS	0.94	0.99	0.99	1.00	1.00	1.00	0.97	0.98
FS	0.90	0.84	0.99	1.00	1.00	1.00	0.97	0.96
CB	0.92	0.77	1.00	1.00	1.00	1.00	0.97	0.95

TABLE II
FINALIZED FEATURE SETS

Domain	**Features**
Registry	Registry handles, open handles details, shimcache, pslist handles, pslist threads
DLLs	DLL load time, DLL path, DLL base, ldr modules, ldr mapped path, ldr base, envars process, envars base, privs process, getsids process
APIs	APIhooks module, APIhooks functions, APIhooks data, threads details, assembly flags, malfind flags
Network	Slack data, destination IP, destination port, source port, flags

D. *Model Training*

Finally, the finalized feature sets were trained with Recurrent Neural Network (RNN) deep learning algorithm using the Keras-TensorFlow python neural network library. The RNN structures are Multi-Layer Perceptron (MLP) with the difference that the input layer is constituted by input neurons and context units, which store delayed hidden layer neurons values from the previous time step to present them to the network as additional inputs in the current time step [16]. The Keras sequential API was used to build the layers for the network. The network consists of layers LSTM cells with dropout to prevent overfitting, fully connected dense layers with relu activation function as hidden layers, a dropout layer to prevent overfitting to the training data and a fully connected dense output layer with softmax function which produces the probability for maliciousness.

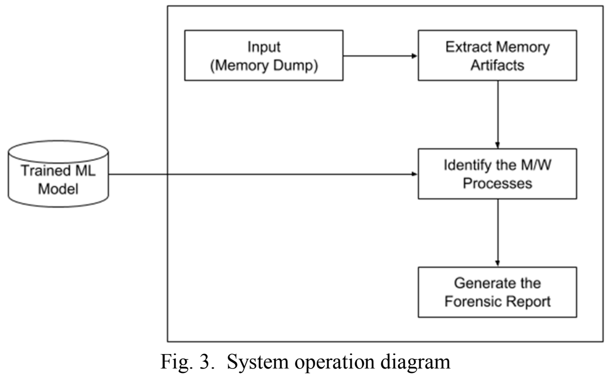


Fig. 3. System operation diagram

IV. RESULTS & DISCUSSION

To obtain the optimized trained model, the neural network was tuned with multiple optimizers (rmsprop, adam), multiple initializers (glorot_uniform, normal, uniform), multiple epochs, multiple batch sizes, multiple loss functions (cross-entropy, squared error, absolute error, squared hinge, hinge) and a range of learning rates, and the optimized set of parameters were identified using GridSearchCV API in Scikit-Learn. Additionally, one-hot encoding the data improved the accuracy. Table III demonstrates the optimized set of parameters for the different domains and the accuracy obtained.

TABLE III
OPTIMIZED SET OF PARAMETERS FOR RNN

Domain	**Epochs**	**Learning Rate**	**Loss Function**	**Optimizer**	**Accuracy**
DLLs	20	0.03	CrossEntropy	rmsprop	0.99
APIs	158	0.001	CrossEntropy	adam	0.98
Network	80	0.04	SquaredError	rmsprop	0.95
Registry	40	0.001	CrossEntropy	adam	0.93

All the above domains behaved well with the ‘uniform’ initializer.

V. CONCLUSION AND FUTURE WORK

According to the security researcher’s suggestion, malware is rapidly incrementing, every second in the digital world. Due to this rapid growth, effective forensic analysis and malware analysis is required. In this paper, we demonstrated an automated system that can automate the entire memory forensic process by predicting the malicious processes existent in the given memory dump. Furthermore, we explored four malware feature types which can be extracted from the memory dump and used to identify the presence of the malicious processes in the system. Moreover, it can be concluded that the DLL artifact plays a major role in predicting malicious processes, using their behavior patterns.

The road ahead includes exploring more memory artifacts and using different methods and factors to analyze malicious processes. Specifically, we plan to classify malware into their family, based on its structure, behavior and impact. Additionally, we plan to analyze the strings patterns on DLL dumps, malfind dumps, process dumps and assembly threads to detect the malicious processes, based on the codes injected and executed to make it more effective.

ACKNOWLEDGMENT

This research is a part of the BSc. (Hons) in Information Technology (specializing in Cyber Security) at SLIIT. The authors would like to express their sincerest gratitude and appreciation for all the researchers who have made the related information, required to complete this research, readily available. Appreciation for SLIIT for providing free online resources, our supervisors for the best guidelines and support throughout the research duration, and the instructors for providing timely support.

REFERENCES

[1] Av-test.org, *Malware Statistics & Trends Report*, AV-TEST-The Independent IT-Security Institute, 2019. Accessed on: Mar. 12, 2019. [Online]. Available at: https://www.av-test.org/en/statistics/malware/

[2] R. Mosli, R. Li, B. Yuan and Y. Pan, “Automated malware detection using artifacts in forensic memory images,” in 2016 IEEE Symp Technologies for Homeland Security (HST), Waltham, MA, 2016, pp.1–6.

[3] SearchSecurity, *What is Antivirus Software (Antivirus Program)?,* WhatIs.com, 2019. Accessed on: Mar. 12, 2019. [Online]. Available at: https://searchsecurity.techtarget.com/definition/antivirus-software

[4] code.google.com, *Volatility Framework*. Accessed on: Mar. 12, 2019. [Online]. Available at: https://code.google.com/p/volatility

[5] A. Aljaedi, D. Lindskog, P. Zavarsky, R. Ruhl and F. Almari, “Comparative Analysis of Volatile Memory Forensics: Live Response vs. Memory Imaging,” in 2011 IEEE Third Int Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third Int Conference on Social Computing, Boston, MA, 2011, pp.1253–1258.

[6] I. Firdausi, C. Lim, A. Erwin and A. S. Nugroho, “Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection,” in 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, Jakarta, 2010, pp.201–203.

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

[8] V. Mee, T. Tryfonas, and I. Sutherland, “The Windows registry as a forensic artefact: Illustrating evidence collection for Internet usage,” *Digital Investigation*, vol. 3, issue 3, pp. 166–173, Sep. 2006.

[9] Y. Duan, X. Fu, B. Luo, Z. Wang, J. Shi and X. Du, “Detective: Automatically identify and analyze malware processes in forensic scenarios via DLLs,” in 2015 IEEE International Conference on Communications (ICC), London, 2015, pp.5691–5696.

[10] Y. Qiao, Y. Yang, L. Ji and J. He, “Analyzing Malware by Abstracting the Frequent Itemsets in API Call Sequences,” in 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, Melbourne, VIC, 2013, pp.265–270.

[11] “*Google Rekall*,” 2013. Accessed on: Mar. 12, 2019. [Online]. Available at: http://www.rekall-forensic

[12] C. Lim and K. Ramli, “Mal-ONE: A unified framework for fast and efficient malware detection,” in 2014 2nd International Conference on Technology, Informatics, Management, Engineering & Environment, Bandung, 2014, pp 1–6.

[13] S. Thomas, K. K. Sherly and S. Dija, “Extraction of memory forensic artifacts from windows 7 RAM image,” in 2013 IEEE Conference on Information & Communication Technologies, Thuckalay, Tamil Nadu, India, 2013, pp.937–942.

[14] C. Rathnayaka and A. Jamdagni, “An Efficient Approach for Advanced Malware Analysis Using Memory Forensic Technique,” in 2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, NSW, 2017, pp.1145–1150.

[15] O. Ferrand, “How to detect the cuckoo sandbox and to strengthen it?,” *Jour Computer Virology and Hacking Techniques*, vol. 11, issue 1, pp. 51–58, Feb. 2015

[16] R. E. Samin, R. M. Kasmani, A. Khamis and S. Isa, “Forecasting Sunspot Numbers with Recurrent Neural Networks (RNN) Using ‘Sunspot Neural Forecaster’ System,” in 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, Jakarta, 2010, pp. 10–14.

[17] Docs.microsoft.com, *Filter Based Feature Selection*, Azure Machine Learning Studio, 2019. Accessed on: Mar. 12, 2019. [Online]. Available at: https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/filter-based-feature-selection

[18] GitHub, *Volatility Foundation/Volatility*, 2019. Accessed on: Mar. 23, 2019. [Online]. Available at: https://github.com/volatilityfoundation/volatility/wiki/Command-Reference#cmdscan