Detection of Linux Malware Using System Tracers – An Overview of Solutions

Igor Vurdelja, Ivan Blažić, Dražen Drašković, Boško Nikolić

***Abstract*— Linux is widely used for servers and embedded systems which require a high level of security and reliability. Although Linux is secure in general, traditional defense methods, such as signature-based detection, fail to detect new malicious programs. A more advanced approach is based on prediction of malicious behavior with dynamic analysis of the executed process. One method of observing the process execution on a Linux system is the use of system tracers such as ftrace and strace. This paper presents an overview of solutions for malware detection by using system tracers on Linux. Different malware detection strategies are discussed and compared with the presented approach. Results of several research projects done in this area are discussed, as well as the observed drawbacks. Technical details of this approach including the tracing utilities on Linux, sandboxing methods and machine learning models are discussed.**

***Index Terms*—Computer Security, Dynamic Analysis, System Tracers, Linux.**

I. Introduction

Computer security is an important topic for many existing infrastructures. With the introduction of Internet of things (IoT), increasing number of devices have network access. These trends may bring additional convenience to everyday life, but also introduce new risks to security and privacy. Different threats can be identified for a particular system, such as DDoS attacks, sniffing attack, SQL injection, XSS, or even social engineering methods. Most of the currently used defense mechanisms are based on anti-virus tools, encryption, authentication, policies, user education, backups, or physical security [1].

Traditional defense mechanisms fail to detect zero-day malware as they are mostly based on identifying programs that are already known to be malicious [2][3]. A more complete defense includes protection against new malware. Accurate detection of new malicious programs is a difficult task due to the sophisticated behavior of malware [4] that hides its malicious functionality, and the number of false positives. This is a classification task that cannot be solved with the same methods as detecting existing malware.

One approach to this problem is the application of machine learning classification algorithms. In this approach a dataset with benign and malicious execution information is used to train a model that detects one or the other type of behavior [5][6][7][8]. System tracers present a universal solution for obtaining the execution information.

Linux is a popular choice for servers [9] and embedded systems [10], because of its benefits in performance, reliability and ease of development. These types of systems are at a higher security risk, as they may be a database server, network equipment, or a control unit of a safety critical device. The above-mentioned approach to detecting zero-day malware is applicable to Linux systems, as there are several system tracers available for this platform.

This paper presents an overview of research focused on malware detection using system tracers on Linux. The following section presents related research projects. Section III presents existing types of malware and detection methods. Section IV presents sandboxing environments for simulating malware execution, and section V presents the available system tracing tools on Linux. Pros and cons of the solutions described in the related research are discussed in section VI. Final section presents the conclusion of this paper.

II. Overview of Related Research

This section contains an overview of several researches of malware detection methods using system traces on Linux. Results of these researches, as well as the applied approaches are presented in this section and further discussed in the section VI.

Authors in [5] proposed an approach based on machine learning technique which uses system calls as features. Malware dataset is obtained from VX-heavens, and it contains 226 malware samples. Used dataset contains 226 malware and 442 benign samples. Results of the experiments presented in this research show a 97% classification accuracy. Malicious samples were executed in a virtual machine in order to collect system calls using strace. After each execution, the virtual machine was restored to clean state. System calls are divided in four categories: union, intersection, discriminating features for malware and discriminating features for benign programs. From each category the features are selected using multiple methods: Class Discrimination Measure (CDM), Odds Ratio (OR), Elimination of Sparse features (ESF). Authors examined how the choice of input data, classifier and feature length would affect the detection accuracy. In this study multiple classifiers are used: Naïve Bayes, J48, AdaboostMl (148), IBKS and Random Forest. It is experimentally shown that best accuracy is achieved with union and intersection sets as input data, Odds Ratio as feature selection method and Random Forest for classification. Accuracy of this model is 97,3%. It is also shown that discriminating features does not provide a good accuracy for distinguishing

Igor Vurdelja is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: vi195024p@student.etf.bg.ac.rs).

Ivan Blažić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: blazic.ivan@outlook.com).

Drazen Draskovic is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: drazen.draskovic@etf.bg.ac.rs).

Bosko Nikolic is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: bosko.nikolic@etf.bg.ac.rs).

malwares from benign programs.

Research [6] was done by same authors as [5] and it is an upgraded version of technique described in the previous work. Authors presented 99,4% accuracy which is even higher than in the previous research. Accuracy was improved by changing the feature selection method. Previously used CDM, OR and ESF have been replaced with eXtended-Symmetric Uncertainty. In X-Symmetric Uncertainty, two-step dimensionality reduction is employed, and the features are ranked based on feature to class correlation and feature to feature inter-correlation. This approach allowed to pick features that had the biggest ability to predict the correct class and to eliminate features that are highly correlated, because they do not bring any useful information about the predicted classes. Best results have been achieved using Random Forest classifier with 27 features selected by eXtended-Symmetric Uncertainty. In this way, the authors managed to make a faster classifier with higher precision compared to their previous work.

Home routers and IoT devices are sensitive to DDoS malware. These devices usually run on Linux. One research [7] examined the application of system call analysis in order to classify behavioral anomalies occurring when the device is infected. Virtual machines running ARM based router firmware were used to simulate the victim device. Traffic simulation was done with a dataset of real anonymized internet traffic and ftrace was used to collect system calls. Two of the most popular DDoS botnet malware types for IoT device were detected, MrBlack and Mirai. These types of malware work by scanning the network for unsecured devices and infecting them by SSH or HTTP authentication with most commonly used credentials. After the device has been infected, it performs DDoS attacks and keeps spreading the malware in the same way. Raw system calls were processed for classification algorithms by extracting *n*-grams of system calls and applying TF-IDF transformation. This is an NLP approach that is commonly used to transform system call logs into machine learning data. Classification was done in different approaches, PCA anomaly detection, one-class SVM classification, naive detection based on identifying the set of *n*-grams that appears in normal traffic. Results of this research presented a 100% accuracy of all three approaches with sufficiently long *n*-grams. The rationale behind such a high accuracy of anomaly detection is the fact that these devices are highly specialized, therefore have strong patterns of normal behavior that are easily broken by malware.

A different approach was taken in a research that represents system calls as *hyper-grams* [8]. In this research the limitations of using *n*-grams for this kind of analysis is mentioned. These limitations mostly come down to observing system calls as raw sequences, without analyzing their individual functionalities and feeding the machine learning algorithms with small raw data that holds no generalized information about the characteristics of process behavior. *Hyper-grams* used in this research are a concise representation of system calls that occurred during a certain period of execution. Every system call is represented in a separate dimension, where the value in each dimension tracks the history of occurrences for that system call. This *hyper-gram* is parametrized with diminishing, addition, and sloping factors, that determinate the importance of new vs old information about specific system calls. These parameters allow a *hyper-gram* to contain a longer or a shorter history of a specific system call occurrence during the execution. For achieving best performance, genetic algorithms were used to optimize these parameters. In this way, a certain period of execution is described as a point in a multidimensional space. Benign and malicious points are identified by the number of their occurrences in benign and malicious processes. In-execution classification proposed in this research is based on re-calculating the point of the observed process with every new system call, and when the average point of this process is dominantly seen in malicious examples, the process is classified as malicious. An experiment was performed to test the approach, where 72 malicious and 72 benign processes were observed on a Linux virtual machine. Results of this experiment presented an AUC of 87,85 with the application of *hyper-grams*. For comparison, the same experiment was done with *n*-grams with different values of *n* and different classification algorithms (JRIP, J48, Naïve Bayes, SVM, and Instance based learner). Maximum AUC of 87,30 was achieved with Naïve Bayes for a 6-gram.

III. MALWARE DETECTION STRATEGIES

Malware detection methods can be separated into different groups based on detection methods and types of program analysis. Common types of malware and detection methods are presented in this section.

A. *Malware identification methods*

The signature-based approach [2] is the most popular and widely used by commercial antivirus software. Signature-based methods rely on patterns extracted from malicious software binary files. This approach has a small error rate, but it cannot deal with simple obfuscation. Behavior-based [3] malware detection techniques observe the behavior of a program to conclude whether it is malicious or not. This approach has a better result in detecting polymorphic malware than a signature-based approach, but has issues detecting packers. The heuristic approach [11] relies on data mining and machine learning techniques to learn the behavior of a malicious program. This is the most modern approach which is still under research. It has the potential to resolve the issues of signature-based and behavior-based methods.

B. *Type of analysis*

During static analysis [12] the executable is analyzed on a file structure bases without execution. As the file is not executed, this type of analysis is fast and simple to deploy. On the other hand, using only static properties of an executable file might not accurately distinguish between benign and malicious executable due to malware's ability to modify or hide the binary code in order to preserve the same malicious behavior. The following techniques are commonly applied to hide the malicious properties of a program:

1) *Metamorphism*

Metamorphic malware will change its code on every execution. This is achieved by replacing existing instructions with a similar one. Despite the permanent changes to code, each iteration of metamorphic malware functions the same way.

2) Polymorphism

Polymorphic malware is a type of malware that changes its shape as well as signatures. It has two parts, but one of them will remain the same with each iteration, so this type of malware is easier to detect.

3) Packing

Packed malware is a type of malware that has been modified using some compression or encryption algorithm. The original executable is compressed, and an unpacked stub is appended to it. When loaded, the unpacked stub will unpack the whole executable and start original malware.

C. Limitations of analysis

Unlike static analysis, dynamic analysis does not include inspection of the binary code, instead, a malicious program is executed in a controlled environment. During this execution, malware will make a trace, or it will make behavior patterns that can be used to detect malicious behavior. The main advantage of dynamic malware analysis is that it is reliable for detecting unknown, metamorphic, and polymorphic malware. The disadvantage of dynamic analysis is that it is neither fast nor safe, and suffers from incomplete code coverage because it monitors only a single execution path. Additional limitation of dynamic analysis is the ability of modern malware to detect a controlled environment.

As static analysis suffers from severe limitations, this paper focuses on dynamic malware analysis using machine learning techniques. In the following chapters, the most common techniques used in dynamic malware analysis are presented.

IV. MALWARE SIMULATION INFRASTRUCTURE

Dynamic malware analysis poses several challenges. The execution of malicious software can damage the host device, or another device connected to the same network. To protect the host from getting infected, a technique called sandboxing is used. This technique is often used by malware analysts to conduct dynamic analysis of untrusted files. A good sandboxing environment fulfills the following properties:

A. Secure isolation

The sandbox is an environment completely isolated from live systems, but it should simulate a live system to ensure that malware will run in the same way as in a real environment. By executing malicious code in a sandbox malware analyst can observe any impact on the potential victim, such as system configuration changes, network calls, or file system changes.

B. Ability to revert the environment to clean state

In automated malware analysis it is important that the system is reverted to a clean state every time new malware is analyzed. If the system is not reverted to a clean state data collected in this way is unreliable as the executed test may compromise the execution of the next test.

C. Realistic simulation of execution environment

Hiding a sandboxed environment consists of hiding virtual environments, as well as any tool used for malware analysis. Authors in [13] have shown that 17% of malware has mechanisms to detect a virtualized environment. Authors in [14] revealed that around 5% of the 110,000 malware samples attempted to evade analysis. Malware has a mechanism to identify whether they are being executed on a real system or a fake one. They show malicious behavior only after some period of time, or after a specific user action. A sandbox should have a mechanism to mimic real user behavior to provoke malicious behavior.

D. Automated execution of multiple samples

Development of a malware analysis method usually consists of executing a big number of malicious program samples. Therefore, an efficient and automated way of extracting traces for each malware has to be applied. Automated execution environments have the ability to uncover artifacts about the malware in a fast manner. There are several open-source tools for isolating malware execution, but they do not meet the mentioned criteria of a good sandboxing environment. On the other hand, these tools can be used for orchestration of multiple virtual machines on which the malware can be executed, and in this way help in automation of extracting malware traces. In the sense of providing a highly controlled environment, sandboxes may be a specific example of virtualization. Usually, sandboxes are implemented using virtual machines and nested virtualization, but authors in [15] also proposed sandboxing using Docker and LXC containers.

Some of the well-known open-source sandbox systems are Cuckoo and Lemon. They are very similar, and both rely on nested virtualization. On the first level of virtualization a virtual machine controls multiple virtual machines on the second level of virtualization. Malware is executed on the nested virtual machines to provide security and isolation. All nested virtual machines are in a virtual network together with a virtual machine on the first level. Fig. 1. Sandbox architecture presents the architecture of the described sandboxing environments.

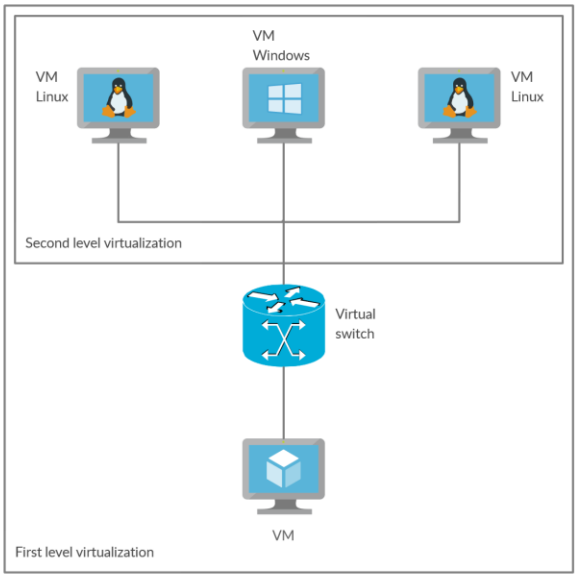

Fig. 1. Sandbox architecture

V. TRACING UTILITIES ON LINUX

System tracers are used to obtain various information about the state of the system or executed processes. In the context of malware detection, data produced by tracers can be used to feed machine learning models that predict malicious behavior or anomalies. Authors in [16][17] showed that system calls can be used with high precision to recognize non-benign behavior.

Linux kernel has an extensive tracing infrastructure useful for debugging, and there are many tools available for system tracing on Linux. These tracers rely on the same mechanisms, but their use cases differ. A brief overview of kernel and user space tracers with tracing mechanisms is presented in Table 1 and Table 2. Most popular kernel tracers are presented in the following text.

TABLE 1
KERNEL TRACERS OVERVIEW

Tool / Tracer	Function tracing	Static tracepoints	Trap	Trampoline
Ftrace	X	X	X	X
LTTng	X		X	X
Perf	X		X	X
eBPF	X		X	X
SystemTap			X	X

TABLE 2
USER SPACE TRACERS OVERVIEW

Tool / Tracer	Function tracing	Static tracepoints	Trap	Trampoline
LTTng	X	X	X	
Printf		X		
Extrae	X			X

Ftrace is a tool that traces Linux kernel internal function calls and interactions between user space and kernel space. This tracer is included in Linux kernel since version 2.6.27. It is a framework of several sub-tracers from which the most typical is the function tracer. Ftrace can be used to trace kernel events like system calls, network traffic, memory access, etc. It relies on several tracing mechanisms including function instrumentation, static tracepoints, and dynamic tracepoints. Ftrace can be enabled or disabled in runtime. Ftrace is manipulated through a set of files in debugfs pseudo-filesystem. Aside from choosing a sub-tracer, ftrace configuration includes setting the size of the trace buffers, and selection of the clock source to use to timestamp the events. Users can choose which events to trace, so the execution overhead of ftrace is small when enabled, and negligible when disabled.

LTTng stands for *Linux Trace Toolkit: next generation*, and it is used for correlated tracing of the user applications, Linux kernel, and user libraries. LTTng dates from 2006, around the same time as ftrace, but LTTng is still not a part of the Linux kernel mainline. It consists of multiple Linux kernel modules for kernel tracing, and dynamically loaded libraries for user space tracing. There are multiple variations of LTTng besides the standard LTTng, these include LTTng-kprobe used for kernel tracing, LTTng UST and LTTng using tracefs for user space tracing. Within the same tracing session, the user can interact with multiple tracers. Most of today's development environments support a graphical interface to inspect LTTng logs.

Perf is a profiler tool for Linux 2.6+ based systems. It serves to monitor the performance of the system. Even though perf uses the same infrastructure as ftrace and LTTng, it is more commonly used as a profiler rather than a tracer. Perf has the ability not only to profile kernel functions but also user space applications. Perf can gather hardware PMU information such as different levels of cache misses, TLB misses, CPU cycles, missed branch predictions, etc. Perf on the other hand is limited to a single process. The events and counters reported by perf are those which occurred within the context of the traced process, and thus have been accounted for it.

SystemTap and eBPF allow users to write programs that can be inserted at run time at any location in kernel using Kprobes. They serve for aggregating and live monitoring of the system rather than for tracing, but can be re-configured to behave like tracers by writing probes that will make samples over time and store them.

Strace is a tool for system calls tracing. It does not come as a part of Linux kernel, so it must be additionally installed. It uses ptrace hooks infrastructure. Once attached to a process it will intercept all its system calls. Strace adds a large overhead, and is commonly used only for testing environments. LTTng and ftrace can achieve the same functionalities as Strace, but they are more lightweight.

VI. ANALYSIS OF THE APPROACH

This chapter presents an overview of the approach to detect malware on Linux using system traces based on existing research. Pros and cons of the methods applied in the previously presented research are discussed.

A. *Selection of the tracing tool*

Picking the suitable tracer is essential to get correct data and to make study expandable in the future. Most of the studies described in section II use strace to extract system calls from program samples.

Strace uses ptrace infrastructure, meaning that malicious file can easily check whether it is under analysis and can change its malicious behavior. Strace is not a standard part of Linux kernel and it must be additionally installed. This can be a pain point for embedded devices which use custom Linux build with Yocto or Buildroot. Strace is limited to tracing system calls only, so no additional information about system state can be obtained. This limits amount of information that could be used for a research. Due to lack of its portability and performance strace would not be suitable for a real-time malware detection tool.

Strace displays system call names and human readable arguments instead of system call numbers which allows an easier platform independent interpretation of system logs. It does not require root privileges and in terms of usability is by far the easiest tracer to be used on Linux. Arguments can be used to enhance the model to be more accurate in detecting malicious binary files. Mentioned limitations of

strace can be overcome by using another tracer such as ftrace.

B. Sandbox environment

Properties of a good sandboxing environment are presented in section IV. Quality of the sandboxing infrastructure can greatly impact the results, but most of the presented papers lack detailed descriptions of these environments.

C. Machine learning applications

A common approach to the classification problem in the related research is machine learning. This kind of approach requires well prepared datasets and an accurate quality evaluation method to be reliable.

All research mentioned in section II suffer from imbalanced or incomplete datasets. Datasets are either small and contain up to 700 samples or are heavily imbalanced containing 80% of malwares and 20% of benign programs. *N*-grams of system calls are usually used as a feature set. When a dataset is small and has a vast number of features the problem of sparse data occurs. In case of *uni*-grams, number of features is around 350, and in case of *bi*-grams, number of features is around 350^2. In these cases classification can be improved by using one-class algorithms, reinforcement learning or obtaining more data.

Malwares are usually distributed as statically linked executables for improved portability, which allows them to infect more devices. On the other hand, benign elf files are usually dynamically linked and taken from `/bin`, `/sbin`, `/usr/bin` or `/usr/sbin`. When executing dynamically linked executables system calls for loading shared libraries can occur. An example of a sequence for loading shared libraries is shown on Fig. 2. Important observation is that the majority of malware does not contain these sequences as they are statically linked, but benign programs do. This can introduce a bias towards classifying statically linked executables as malware, and dynamically linked as benign, which could explain a very high accuracy in previously presented research. Two ways of overcoming this problem are compiling Linux commands as statically linked executables or filtering sequences related to loading shared libraries.

```
open
fcntl
fstat
read
mmap
mmap
mmap
mmap
close
open
fcntl
fstat
read
mmap
mmap
mmap
mmap
close
mprotect
mprotect
mprotect
mprotect
```

Fig. 2. System call sequence for loading shared libraries

Linux commands are used as benign samples in most of the presented research, but criteria for choosing individual commands has not been described. Most of the Linux commands access the file system, but malicious samples usually make significantly more network calls, which can lead to inaccurately biased models. Vast majority of benign Linux commands print on the console while malicious files do not, which can lead to a similar bias problem. System calls used for obtaining information about the console is shown on Fig. 3.

```
ioctl(1, TIOCGWINSZ, {ws_row=36, ws_col=101, ws_xpixel=0, ws_ypixel=0}) = 0
ioctl(1, TIOCGWINSZ, {ws_row=36, ws_col=101, ws_xpixel=0, ws_ypixel=0}) = 0
```

Fig. 3. System calls for getting information about console

System call sequences made by both benign and malicious samples should be as diverse as possible in order to avoid this kind of bias. This can be achieved by a proper selection of benign samples.

An NLP approach has been mostly used, where system trace entries are treated as a set of individual smaller sequences which are then processed with *bag-of-words* algorithms. The NLP approach is considered to be robust in general but could lead to a less reliable classification if normalization such as TF-IDF is not applied or the size of *n*-grams is not optimal. In the context of malware detection, this approach can identify smaller patterns of execution that can frequently appear only in malicious programs. This approach lacks complete information about the execution since the individual sequences of a program are not independent. Data representation using feature vectors such as *hyper-grams* overcomes this limitations to a certain extent, but does not seem to provide significant improvement over *n*-grams [8]. High classification accuracy is presented in the related research, where most results show over 90%. This amount of accuracy is mathematically correct, but may not reflect the predictive power the classifier has in reality [18]. In cases where small or unbalanced datasets are used, a high accuracy will give a false measure of quality.

VII. Conclusion

Analysis of Linux system trace logs for malware classification has been a topic of several research presented in this paper. Approaches presented in the related research consist of executing benign and malware programs in a controlled environment to obtain system trace logs, which are then processed with machine learning algorithms to implement a classifier. Most results of these research present high classification accuracies but may lack appropriate datasets and real predictive power. From a critical analysis of the related research, it has been concluded that this approach requires a bigger, more well-prepared machine learning dataset, and a realistic simulation of the execution. Presented approach to malware detection has a lot of further research potential, with the ultimate goal of having robust real-time detection of zero-day malware on Linux systems.

REFERENCES

[1] Lee Brotherston, Amanda Berlin, *Defensive Security Handbook: Best Practices for Securing Infrastructure*, Newton, Massachusetts, US, O'Reilly Media, Inc, 2017.

[2] P. Gutmann. "The Commercial Malware Industry", available at https://www.cs.auckland.ac.nz/~pgut001/pubs/malware_biz.pdf

[3] W. Liu, P. Ren, K. Liu, H. Duan, "Behavior-Based Malware Analysis and Detection", First International Workshop on Complexity and Data Mining, NW Washington, DC, United States, pp. 39-42, 2011.

[4] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, Davide Balzarotti, "Understanding Linux Malware", IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, pp. 161-175, 2018.

[5] K. A. Asmitha, P. Vinod, "A Machine Learning Approach for Linux Malware Detection", International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), Ghaziabad, India, pp. 825-830, 2014.

[6] K. A. Asmitha, P. Vinod, "Linux Malware Detection Using eXtended-Symmetric Uncertainty", International Conference on Security, Privacy, and Applied Cryptography Engineering, Pune, India, pp. 319-332, 2014.

[7] N. An, A. Duff, G. Naik, M. Faloutsos, S. Weber and S. Mancoridis, "Behavioral anomaly detection of malware on home routers," 12th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, Puerto Rico, United States of America, 2017.

[8] B. Mehdi, F. Ahmed, S. A. Khayyam, M. Farooq, "Towards a Theory of Generalizing System Call Representation for In-Execution Malware Detection", IEEE International Conference on Communications, Cape Town, South Africa, 2010.

[9] "Usage Statistics and Market Share of Operating Systems for Websites, August 2020", W3Techs, accessed 09.09.2020, https://w3techs.com/technologies/overview/operating_system.

[10] "2019 Embedded Markets Study", AspenCore, accessed 09.09.2020, https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[11] Z. Bazrafshan, H. Hashemi, S. Mehdi Hazrati Fard, A. Hamzeh, "A survey on heuristic malware detection techniques", 5th Conference on Information and Knowledge Technology, Shiraz, Iran, pp. 113-120, 2013.

[12] M. G. Schultz, E. Eskin, E. Zadok, S. J. Stolfo, "Data mining methods for detection of new malicious executables", Proceedings of IEEE Symposium on Security and Privacy (S&P), Oakland, California, pp 38-49, 2001.

[13] "Carbank Malware – Ninety Five Percent Exhibits Stealthy or Evasive Behaviors", available at https://www.lastline.com/labsblog/carbanak-malware-ninety-five-percent-exhibits-stealthy-or-evasive-behaviors/, date accessed 8. June 2020.

[14] D. Kirat, G. Vigna, C. Kruegel, "BareCloud: Bare-metal Analysis-based Evasive Malware Detection", USENIX Security Symposium, San Diego, California, pp. 287-301, 2014.

[15] D. Hellinger, L. M. Xuan, P. Gahlot, "Dynamic Analysis of Evasive Malware with a Linux Container Sandbox", available at https://www.researchgate.net/publication/330500642_Dynamic_Analysis_of_Evasive_Malware_with_a_Linux_Container_Sandbox

[16] S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff, "A sense of self for Unix processes", IEEE Symposium on Security and Privacy, Oakland, California, pp. 120-128, 1996.

[17] K. Denney, C, Kaygusuz, J. Zuluaga, "A Survey of Malware Detection Using System Call Tracing Techniques", 2018.

[18] F. J Valverde-Albacete, C. Peláez-Moreno "100% Classification Accuracy Considered Harmful: The Normalized Information Transfer Factor Explains the Accuracy Paradox", PloS one, 2014