Original scientific paper

An Automated Framework for Runtime Analysis of Malicious Executables on Linux

Igor Vurdelja, Ivan Blažić, Dragan Bojić, and Dražen Drašković

***Abstract* — One way of testing a malware detection tool is to expose it to a large number of diverse malware samples and verify its detection accuracy. During these tests, the host system must not be harmed by malware and yet be able to analyze its harmful behavior. An additional challenge is to run a large number of executables in the shortest amount of time. Advanced malware can even stop its execution when detecting a simulation. This paper presents a framework for automated malware analysis on Linux. The proposed solution addresses these problems with parallel realistic simulations. Existing malware analysis methods are discussed, as well as technical details behind reliable execution simulations.**

***Keywords* — Computer Security, Dynamic Analysis, Sandbox, Linux.**

I. Introduction

COMPUTER security is a continuously developing field due to new malware and attack methods evolving each day. Dynamic malware analysis, that is based on observing malicious software execution includes a risk of damaging the test environment itself. Sandboxing [1] is used to ensure proper analysis and protection of the host from the harmful effects of malware, commonly used by commercial anti-virus software. It provides protection by running a piece of software in an environment isolated from the rest of the system, usually achieved with virtualization or physical separation. The development of malware detection technology requires an analysis of common properties of malicious programs that are then recognized during active protection. To perform this kind of analysis, a large number of diverse malware samples need to be executed, which requires a safe and automated execution infrastructure. More advanced malware can bypass detection by hiding its malicious properties when executed in a sandbox. To overcome this problem, the sandbox environment should be difficult to differentiate from a real system. This paper presents a safe automated framework for analyzing large numbers of malware for Linux-based operating systems. The proposed solution is based on the control of parallel running virtual machines with protection from sandbox or virtualization detection. The following chapter presents an overview of existing Solutions. Section III describes sandboxing techniques. Section IV presents malware evasion methods, and section V presents techniques used to counter malware evasion methods. Section VI describes the infrastructure used to extract system logs from malware samples. The final chapter presents the conclusion of the paper.

Paper received April 11, 2021; accepted December 27, 2021. Date of publication December 29, 2021. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Miroslav Lutovac.

This paper is a revised and expanded version of the paper presented at the 28th Telecommunications Forum TELFOR 2020 [2].

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).

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

Dražen Drašković 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).

II. Overview of Existing Solutions

Cuckoo sandbox [3] is an open-source framework for automated malware analysis. It is compatible with a variety of virtualization software that can run virtual machines with most operating systems. The main components of the *Cuckoo* framework are:

1) *Cuckoo* host - used to orchestrate the execution and analysis flow.

2) Analysis guest (agent) - runs in the isolated environment where the malware is executed and reports the results to the host.

Both host and guest can be real or virtual machines. The *Cuckoo* framework does not provide any preparation of the guest machine for purposes of safe analysis, this is up to the user instead. An analysis is performed using packages - python scripts that describe how Cuckoo should conduct the analysis procedure for a given type of file. By default, it supports several file types including executable files for Windows (PE) and Linux (ELF).

The system described in this paper leveraged *Cuckoo*'s modular design to integrate it into a larger framework. In comparison to typical *Cuckoo* applications, the victim virtual machines are designed specifically to run Linux malware with protection against virtualization detection issues described in chapter IV. Additionally, our system uses *ftrace* instead of *systemtap* to extract behavioral traces of malware under test.

It is worth noting that there are websites that allow users to upload files for malware analysis [4], obviating the need for dedicated hardware and sandbox systems. However, this method is limited to simple security analysis of files and does not provide protection against hidden malware behavior that more advanced sandbox systems can detect.

Authors in [5] applied the *Cuckoo* sandbox to design and develop a distributed firewall (*Distfw*) for Linux operating system. IP tables are used to control incoming and outgoing network traffic, and *IPsec* is used for securing network communication. *Cuckoo* is used to automate the process of submitting and evaluating malware samples. Compared to the presented in this paper, *Distwf* is focused only on malware that uses network communication to infect other machines. Additionally, *Distfw* is not resistant to malware evasion techniques.

Studies in [6], [7] focus on the application of *Docker* and LXC containers as a base in the sandboxed environment. Experimental results have shown that containers, same as virtual machines, are subject to fingerprinting through different sources that malware can easily find. This research was focused on ways to make containers less vulnerable. The solution proposed in this research lacks the ability to automatically analyze malware samples. Container-based virtualization offers a lighter, faster, more scalable, but less secure option compared to hypervisor-based sandboxes. Comparison between different sandboxing solutions is described in more detail in the next chapter.

III. Sandboxing Techniques

Sandboxing can be implemented with virtual machines or containerization. In this chapter, an attack surface of containers and virtual machines with type 2 hypervisors is analyzed. An attack surface defines points from which a system can be attacked. Type 1 hypervisors are not considered due to the lack of available hardware for this study. A virtual machine's attack surface consists of exploits in the hypervisor, which would allow one virtual machine to read the memory of another one. Containers have a bit larger attack surface which consists of using the exploits in the container engine (e.g. Docker daemon) and the host operating system. As shown in Fig. 1, every virtual machine has its own operating system, so in case of using the exploit in the operating system, it will impact only a single virtual machine.

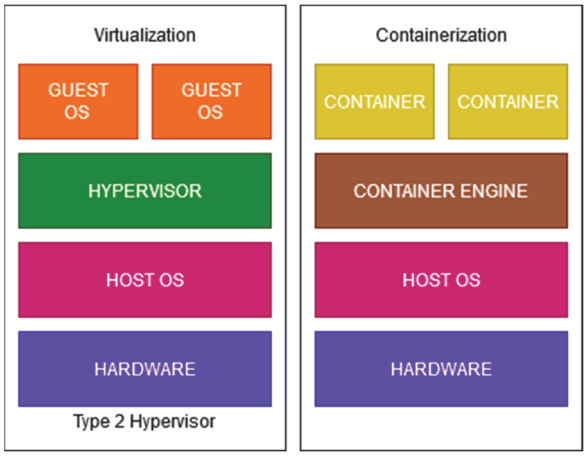

Fig. 1. Comparison between type 2 hypervisor and containerization.

Containers on the other hand isolate data and applications only at a process level. As mentioned in [8] and [9] all containers share the same operating system with the host, meaning that any exploit in the operating system would result in a compromised host machine. According to [10] in the period from 2017 to 2020, 801 Linux kernel exploits were found. Kernel exploits can be devastating for containerized environments, so in this study virtual machines are used. In this research, Virtual Box was used as desktop virtualization software.

An alternative method to virtualization is the use of real hardware. This hardware would be dedicated to malware execution and suffer the consequences of harmful effects. Besides the cost of additional hardware, the biggest drawback of this approach is the difficulty of recovering the machine to a healthy state after malware execution. Virtual machine state can be easily stored and recovered with snapshots, while real hardware would require more effort to restore, for example re-installing the operating system after more severe damage. With a proper configuration, more security can be achieved with this approach due to the separation of the physical layer [11]. Due to the mentioned downsides of using real hardware, this research relies on virtualization.

IV. Malware Evasion Methods

Malware analysis and detection techniques have become known over time. As a consequence, malware authors deploy anti-detection techniques [12] such as virtual machine and container detection, debugger, disassembler, and sandbox detection. The goal of such techniques is to detect whether a malware is being executed in a monitored environment and therefore decide to remain hidden or attack the host. A malware that manages to evade detection is likely to stay under the radar for quite a long time, representing a huge risk. This chapter describes common evasion techniques used by malware developers to detect analysis and hide their actual malicious behavior.

A. Anti-VM techniques

Anti-VM techniques are divided into instruction-based and file-based. As Linux stores system information as files, running it in a virtualized environment implies that files specific to a virtualization tool can be found on the system.

File-based techniques, therefore, rely on finding and accessing system information files. In that way, malware can collect enough information to sense a virtualized environment. Files that are specific to virtualization tools include hardware parameters such as system, bios, board vendor, serial number and, MAC address. Other traces of virtualization can be the hypervisor flag in `/proc/cpuinfo` or the BIOS brand and version.

Instruction-based techniques rely on using a specific set of CPU instructions to provoke those that are different on virtual machines and real hardware. These techniques are more reliable for processors that do not support hardware-assisted virtualization. Without hardware-assisted virtualization discrepancy between virtualized and non-virtualized environments is bigger due to techniques such as binary translation [13] and paravirtualization which is easier to detect. Another type of VM detection is time-based detection, commonly used to detect hardware-assisted virtualization. Time-based detection [14] relies on instructions that take longer to execute in a virtualized

environment than on a real system. Under specific circumstances, such as page faults, the virtual machine returns control to the hypervisor when executing certain instructions, and thus occurs the *vmexit* instruction. Unlike running in the real system, execution in the virtual machine takes longer due to the *vmexit* and *vmresume* instructions.

Anti-VM technology is most commonly found in widely deployed malware, such as bots, intimidation software, and spyware because they are targeting average users which are unlikely to use a virtual machine in their daily work.

B. Anti-sandbox techniques

Anti-sandbox techniques are based on characteristics that are common for almost all sandboxes or a specific sandbox.

Identification strategies based on common characteristics rely on detecting analysis and debugging tools used to conduct dynamic execution analysis. Presence of these tools can be checked by scanning common installation paths in Linux: `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`. To confirm whether these tools are running it is sufficient to check the names of all processes present in the `/proc` pseudo filesystem. Detection of debuggers, e.g., *gdb* or *strace* can be done by calling the *ptrace* system call recursively, because a process can only be traced by one tracer at a time. If the traced program makes a call to *ptrace*, it will return error code -1. An additional way to detect debuggers is by checking the TracerPid field in `/proc/self/status` file. Sandboxed environments usually have limited resources, like hard disk, RAM, and CPU cores. Also, network traffic is usually limited or regulated by IP tables. By checking these resources in the execution environment, malware can detect if it is a sandbox.

Certain identification strategies are based on characteristics of a specific sandbox implementation. Malware developers will try to detect a specific sandbox by examining ports, files, or libraries known to be used by that sandbox. One example is the *Anticuckoo* project [15] which uses techniques to uniquely identify and crash *Cuckoo's* sandbox environment [16].

C. Anti-automated analysis techniques

Anti-automated analysis techniques rely on a lack of human interaction. Automated environments are devoid of human inputs such as the use of a keyboard and mouse. Malware can monitor all events occurring in the Linux input subsystem. If no event is made over some period of time, it can indicate an automated execution. Usually, automated environments monitor malware's execution only for a couple of minutes, which can easily be bypassed by stalling the execution. Another way to bypass automated analysis is to request user interactions such as GUI or console prompts.

V. Countering Malware Evasion Methods

In this chapter, different measures applied to hide simulated environments are presented. Techniques that will not be covered are related to anti-VM techniques described in [17] and [18]. VirtualBox was used as a virtualization software in this research. All exploits in the Virtual Box Hypervisor that cause it to emulate CPU instructions differently than on a real machine will not be considered, as there is no reliable defense against these types of bugs.

In a newly created virtual machine, VirtualBox will set BIOS, system, board, chassis, processor, OEM, primary and secondary IDE to VirtualBox-specific names. The first step in hiding a virtualized environment is to anonymize the hardware identifiers. Since VirtualBox version 4.2 this can be achieved with the *vboxmanage* command. The sample code used to modify bios information of the virtual machine called cuckoo_victim is shown in Fig. 2. A good practice would be to change the original hardware information with the real hardware information of the host computer. With this modification, the behavior will be more close to a real machine. To implement hardware anonymization, several scripts for Linux and Windows are created within this research.

```
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSVendor" "Acer"
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSVersion" "Acer"
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseDate" "25/02/2010"
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseMajor" 2
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseMinor" 3
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSFirmwareMajor" 5
/usr/bin/vboxmanage setextradata cuckoo_victim \
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSFirmwareMinor" 7
```

Fig. 2. Commands to modify virtual machine's BIOS information.

By the default VirtualBox VM settings, the MAC address will correlate to an Oracle network card. In order to avoid exposure through analysis of the network card, the MAC address prefix should be changed. This can be done with the *vboxmanage* command or VirtualBox GUI.

The paravirtual interface should be set to *None*. Disabling the paravirtualization interface will make the detection of hypervisor more difficult. Malware will not be capable of getting the hypervisor information through the *cpuid* instruction.

VirtualBox guest additions plugin is used to provide better interaction between the host machine, e.g., easier file-sharing or image scaling. Guest additions are useful for the typical use of virtual machines but should be avoided in malware analysis applications due to additional kernel modules which can easily be detected.

An important step in hiding a simulated environment is to hide any tool used for extracting data from malware. In this study, no debugger or tracer that requires *ptrace* infrastructure was used, so there was no need to hide the tracer information. For extracting information about malware *ftrace* [19] was used, which is silent and doesn't set TracerPid in `/proc/self/status` therefore cannot be detected by a recursive call to *ptrace*. All tools used for data extraction should be renamed, as most malware will check if common analysis tools are present on the system.

Malware can be designed to detect specifically the *Cuckoo* process on the system by searching for the process named “agent.py” which is commonly used by the *Cuckoo* sandbox. For protection, it is sufficient to change the name of the python script. Malware may check if python is present and running on the system. To avoid this, the python script can be converted to the standalone executable using *pyinstaller*.

As mentioned in the previous chapter, an automated environment can be detected by the lack of keyboard and

mouse input events. This was solved by generating input events and writing them directly in Linux input subsystem, e.g., `/dev/input/event0`.

When creating a virtual machine at least 2GB of RAM and 10GB of disk memory should be allocated to mimic the amount of resources available on a real system. If possible, at least 2 CPU cores should be allocated to the virtual machine.

After applying the previously described methods, it is necessary to check whether they are effective. As this process is tedious and time-consuming, an open-source tool [20] has been developed to improve time efficiency. It is completely implemented in C++ and employs techniques (described in chapter IV) to analyze the environment the same way a Linux malware would do, but is completely benign. This tool can help measure how well the virtualized environment is hidden. To demonstrate this tool, the output of the program was compared when running without and from *gdb*, as shown in Fig. 3. Output of the test program used for checking the quality of the victim machine. On the left picture, test program is started without *gdb* compared to the right picture where the program is started with *gdb*.The tool managed to recognize that *gdb* is running on the system and that *gdb* is used to trace its behavior. Both times the tool was started in the user virtual machine which was not prepared for malware analysis.

```
Testing sandboxed environment
Checking:
If traced via proc fs:      Not detected!
If traced via ptrace:       Not detected!
For ptrace monkey patch:    Not detected!
Active analysis tools:      Not detected!
Installed analysis tools:   Detected!
For common program names:   Not detected!
Is program name md5 hash:   Not detected!
Number of cpu cores:        Not detected!
Total disk size:            Not detected!
Available disk size:        Not detected!
Is sleep accelerated via:
 - sysinfo:                 Not detected!
 - proc fs:                 Not detected!

Testing automated analysis
Checking:
Input events:               Detected!
For timeouts:               Not detected!

Testing virtualized environment
Checking:
Bios vendor:                Detected!
Product vendor:             Detected!
System vendor:              Detected!
Board vendor:               Detected!
Kernel modules:             Detected!
SCSI:                       Detected!
Hypervisor flag:            Detected!
VM presence:                Not detected!
MAC address:                Detected!
Hypervisor bit:             Detected!
Time of VMEXIT:             Not detected!
Virtualization vendor:      Detected!
IN instruction:             Not detected!
```

```
Testing sandboxed environment
Checking:
If traced via proc fs:      Detected!
If traced via ptrace:       Detected!
For ptrace monkey patch:    Not detected!
Active analysis tools:      Detected!
Installed analysis tools:   Detected!
For common program names:   Not detected!
Is program name md5 hash:   Not detected!
Number of cpu cores:        Not detected!
Total disk size:            Not detected!
Available disk size:        Not detected!
Is sleep accelerated via:
 - sysinfo:                 Not detected!
 - proc fs:                 Not detected!

Testing automated analysis
Checking:
Input events:               Detected!
For timeouts:               Not detected!

Testing virtualized environment
Checking:
Bios vendor:                Detected!
Product vendor:             Detected!
System vendor:              Detected!
Board vendor:               Detected!
Kernel modules:             Detected!
SCSI:                       Detected!
Hypervisor flag:            Detected!
VM presence:                Not detected!
MAC address:                Detected!
Hypervisor bit:             Detected!
Time of VMEXIT:             Detected!
Virtualization vendor:      Detected!
IN instruction:             Not detected!
```

Fig. 3. Output of the test program used for checking the quality of the victim machine. On the left picture, test program is started without *gdb* compared to the right picture where the program is started with *gdb*.

VI. Framework Implementation

This chapter describes the framework for a reliable automated malware analysis developed within this research. The proposed solution addresses the virtualization and sandboxing detection problems mentioned in the previous chapters, as well as the performance issues with executions of a large number of malware samples.

Nested virtualization is used as the main approach to protecting the test environment and the host machine. The first level of virtualization runs a Ubuntu Linux VM used for the orchestration of victim machines and the collection of execution logs. The second level of virtualization contains up to five lightweight Alpine Linux VMs used for the actual malware execution. An overview of the architecture is presented in Fig. 4. The framework is configurable so that the user can choose the number of victim VMs that run in parallel.

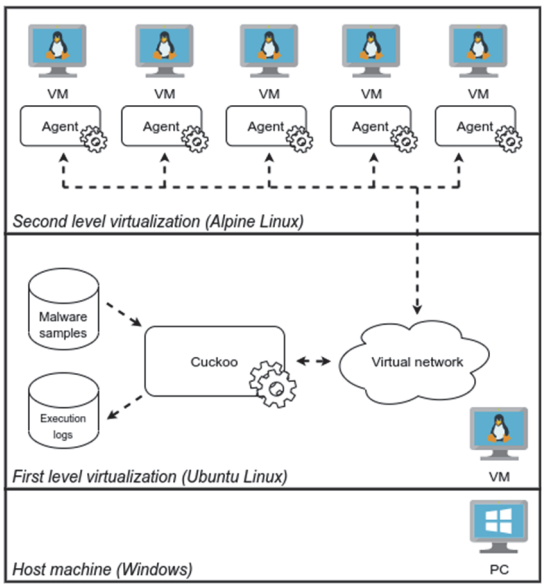

Fig. 4. System architecture.

Nested virtual machines are protected from malware evasion techniques by methods described in chapter IV. The *Cuckoo* framework was used for orchestrating nested virtual machines. Features of the Cuckoo framework include communication between the host and the victim machines, restoring snapshots, and gathering system logs. Communication is done via TCP sockets, where the *Cuckoo* process on the host machine communicates with several agent processes, which are running on the victim machines. An agent process receives the malware sample, starts *ftrace*, runs the malware, and finishes the analysis. The analysis is done when the malware sample finished its execution or when the analysis timeout expires. In order to prevent malware from reaching anything outside the sandbox, the host and the victim machines are placed in an isolated virtual network. Additionally, the host machine is protected by IP tables rules which block any traffic not generated by the *Cuckoo* framework.

The presented framework is used with a simple python command-line interface. *Ftrace* data gathered from execution tracing is in the form of raw logs. This raw data needs to be processed to extract meaningful information about the execution or transform it into a more useful form. The presented framework contains a data processing tool that provides these operations.

Inputs for the data processing tool are the *ftrace* files gathered from the execution, and the supported operations are:

- Extraction of the system call occurrences
- Verification and cleanup of raw data
- Statistical analysis of the system call occurrences

The raw dataset contains a large number of *ftrace* logs, and each log needs to correspond to a single executable that will later be classified as benign or malicious.

Verification and cleanup of this data ensure that there are no duplicate entries, missing log files, or wrong correspondence between the executable and the execution log.

When the data is cleaned up, the tool parses log files to extract system call occurrences. These occurrences are presented in their original sequence with system call names instead of identifiers. These extracted system call sequences are suitable for machine learning datasets. Since the original sequence is preserved, machine learning algorithms can take advantage of both the number of occurrences and their order.

Statistical analysis of the system call occurrences involves the calculation of dataset properties such as the number or specific system calls in benign or malicious samples, system calls that occur in one category only, the total number of analyzed samples, etc. This statistical data is not suitable for machine learning, but instead for having a comparison of system calls in malicious and benign programs.

The proposed solution may suffer from performance and stability issues due to the number of virtual machines running in parallel. This could be resolved with an alternative approach where each victim virtual machine runs on dedicated hardware. Another drawback is the vulnerability to potential hypervisor bugs mentioned in the previous chapter.

VII. Conclusion

Sandboxing is a common approach to runtime malware detection. This paper presents a Linux framework for automated malware analysis based on the extraction of information about malware executed in a sandbox environment. Multiple approaches to creating a safe sandbox environment were analyzed, where virtualization was considered the best option. Techniques performed by malware to evade detection are discussed and addressed in the solution. The implemented framework could be used for developing a novel machine learning-based solution for detecting zero-day malware. Future work would include improvements in the performance and scalability of the framework.

References

[1] I. Vurdelja, I. Blažić, D. Drašković, B. Nikolić, "Detection of Linux Malware Using System Tracers – An Overview of Solutions", in *Proceedings of the 7th International Conference on Electrical, Electronic and Computing Engineering - IcEtran 2020*, pp. 597- 602, Sept. 2020.

[2] I. Vurdelja, I. Blažić, D. Bojić, D. Drašković, "A framework for automated dynamic malware analysis for Linux", in *Proceedings of the 28th Telecommunications Forum - TELFOR 2020*, IEEE, Belgrade, Serbia, pp. 1-4, Nov. 2020, DOI: https://doi.org/10.1109/TELFOR51502.2020.9306520.

[3] D. Oktavianto, I. Muhardianto, *Cuckoo Malware Analysis*, Packt Publishing, 2013.

[4] Cuckoo online analysis, available at https://cuckoo.cert.ee/, date accessed 11.4.2021.

[5] M. Vasilescu, L. Gheorghe, N. Tapus, "Practical malware analysis based on sandboxing", in *2014 RoEduNet Conference 13th Edition: Networking in Education and Research JointEvent RENAM 8th Conference*, Chisinau, Moldova, pp. 1–6, Sept 2014, DOI: https://doi.org/10.1109/RoEduNet-RENAM.2014.6955304.

[6] A. Khalimov, S. Benahmed, R. Hussain, S.M. A. Kazmi, A. Oracevic, F. Hussain, F. Ahmad, C. A. Kerrache, "Container-based Sandboxes for Malware Analysis: A Compromise Worth Considering", in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'19), Association for Computing Machinery*, New York, NY, USA, pp. 219–227, Dec 2019, DOI: https://doi.org/10.1145/3344341.3368810.

[7] 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.

[8] S. Sultan, I. Ahmad and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead", in *IEEE Access*, vol. 7, pp. 52976-52996, Apr. 2019, DOI: https://doi.org/10.1109/ACCESS.2019.2911732.

[9] R. Mehra, "Docker Containers Versus Virtual Machine-Based Virtualization", in *Proceedings of the international conference on emerging technologies in data mining and information security (IEMIS)*, Kolkata, India, vol. 3, pp. 141-150, Feb. 2018, DOI: https://doi.org/10.1007/978-981-13-1501-5_12.

[10] "CVE details", available at https://www.cvedetails.com/, date accessed 11. April 2021.

[11] Chad Spensky, Hongyi Hu, Kevin Leach, "LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis", in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, pp. 1-15, Feb. 2016.

[12] A. Afianian, S. Niksefat, B. Sadeghiyan, D. Baptiste, "Malware Dynamic Analysis Evasion Techniques: A Survey", in *ACM Computing Surveys*, vol. 52, no. 6, pp. 1-28, Nov. 2019, DOI: https://doi.org/10.1145/3365001.

[13] B. Rodrigues, F. Cerveira, R. Barbosa, J. Bernardino, "Virtualization: Past and Present Challenges", in *Proceedings of the 13th International Conference on Software Technologies (ICSOFT)*, Porto, Portugal, pp. 755-761, Jul. 2018, DOI: https://doi.org/10.5220/0006910707890795.

[14] I. Korkin, "Two Challenges of Stealthy Hypervisors Detection: Time Cheating and Data Fluctuations", in *Annual Conference on Digital Forensics, Security and Law (CDFSL)*, Daytona Beach, Florida, USA, pp. 33-57, May 2015.

[15] "Anticuckoo", available at https://github.com/David-Reguera-Garcia-Dreg/anticuckoo, date accessed 11. April 2021.

[16] Cuckoo sandbox, available at https://cuckoosandbox.org, date accessed 11. April 2021.

[17] R. Paleari, L. Martignoni, G. F. Roglia, D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect CPU emulators", in *USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada, Aug. 2009.

[18] H. Shi, J. Mirkovic, A. Alwabel, "Handling Anti-Virtual Machine Techniques in Malicious Software", in *ACM Transaction on Privacy and Security*, vol. 21, no. 1, article no. 2, pp. 1-31, December 2017, DOI: https://doi.org/10.1145/3139292

[19] Ftrace, available at https://elinux.org/Ftrace, date accessed 11. Apr. 2021.

[20] Igor Vurdelja, mlw_anti_analysis, GitHub repository, (2021) https://github.com/vurdeljica/mlw_anti_analysis