





Article

Enhancing Linux System Security: A Kernel-Based Approach to Fileless Malware Detection and Mitigation

Min-Hao Wu ¹, Fu-Hau Hsu ^{2,*}, Jian-Hung Huang ², Keyuan Wang ², Yan-Ling Hwang ³, Hao-Jyun Wang ², Jian-Xin Chen ², Teng-Chuan Hsiao ² and Hao-Tsung Yang ²

- ¹ College of Artificial Intelligence, Xiamen City University, Xiamen 361000, China; mhwu@csie.ncu.edu.tw
- ² Department of Computer Science and Information Engineering, National Central University, Taoyuan 32001, Taiwan; 109522059@cc.ncu.edu.tw (J.-H.H.); sixkwnp@ee.ncu.edu.tw (K.W.); alan.wang388@g.ncu.edu.tw (H.-J.W.); opp556687@g.ncu.edu.tw (J.-X.C.); s29912127@gmail.com (T.-C.H.); htyang@ncu.edu.tw (H.-T.Y.)
- ³ Department of Applied Foreign Languages, Chung Shan Medical University, Taichung 40201, Taiwan; yanling@csmu.edu.tw
- * Correspondence: hsfh@csie.ncu.edu.tw

Abstract: In the late 20th century, computer viruses emerged as powerful malware that resides permanently in target hosts. For a virus to function, it must load into memory from persistent storage, such as a file on a hard drive. Due to the significant destructive potential of viruses, numerous defense measures have been developed to protect computer systems. Among these, antivirus software is one of the most recognized and widely used. Typically, antivirus solutions rely on static analysis (signature-based) technologies to detect infections in files stored on permanent storage devices, such as hard drives or USB (Universal Serial Bus) flash drives. However, a new breed of malware, fileless malware, has been designed to evade detection and enhance durability. Fileless malware resides solely in the memory of the target hosts, circumventing traditional antivirus software, which cannot access or analyze processes executed directly from memory. This study proposes the Check-on-Execution (CoE) kernel-based approach to detect fileless malware on Linux systems. CoE intervenes by suspending code execution before a program executes code from a process's writable and executable memory area. To prevent the execution of fileless malware, CoE extracts the code from memory, packages it with an ELF (Executable and Linkable Format) header to create an ELF file, and uses VirusTotal for analysis. Experimental results demonstrate that CoE significantly enhances a Linux system's ability to defend against fileless malware. Additionally, CoE effectively protects against shell code injection attacks, including buffer and memory overflows, and can handle packed malware. However, it is important to note that this study focuses exclusively on fileless malware, and further research is needed to address other types of malware.



Citation: Wu, M.-H.; Hsu, F.-H.; Huang, J.-H.; Wang, K.; Hwang, Y.-L.; Wang, H.-J.; Chen, J.-X.; Hsiao, T.-C.; Yang, H.-T. Enhancing Linux System Security: A Kernel-Based Approach to Fileless Malware Detection and Mitigation. *Electronics* **2024**, *13*, 3569. <https://doi.org/10.3390/electronics13173569>

Academic Editors: Aryya Gangopadhyay and Rajendra V. Boppana

Received: 1 August 2024
Revised: 4 September 2024
Accepted: 6 September 2024
Published: 8 September 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: antivirus; fileless malware; dynamic analysis; memory analysis

1. Introduction

The proposed framework for detecting and mitigating fileless malware in Linux systems has a significant relationship with the field of electronics, particularly in the context of Internet of Things (IoT) devices. Linux is a prevalent operating system for IoT devices due to its flexibility and open-source nature, making it a critical target for malicious attacks. Fileless malware poses a unique threat to these devices, which often have limited computational resources and rely on allow-lists rather than comprehensive antivirus solutions. By enhancing security at the kernel level, the framework addresses the vulnerabilities in electronic systems that use Linux. Integrating such security measures into IoT devices can help safeguard against unauthorized access and manipulation, ensuring the reliability and integrity of electronic systems increasingly interconnected in modern technology landscapes. This relationship highlights the importance of developing advanced

security protocols tailored to the specific needs of electronic systems, thereby contributing to the broader field of electronics cybersecurity. Antivirus software is becoming a crucial defense tool for computer and network systems. Tests comparing antivirus and different viruses demonstrate that antivirus software shields computer systems against malware attacks. Malware software is terminated, and the system is alerted when an antivirus engine finds it. Each side in the continuous conflict between antivirus and malware develops new strategies to trick the other. Finding a means to avoid antivirus detection is one of the efficient tactics malware developers use.

Put another way, they attempt to make their malware invisible to antivirus. Unlike antivirus engines, allow-lists are often used by IoT devices to prevent malware execution due to the constrained resources of IoT. Linux is commonly used on IoT devices. However, fileless malware can be executed undetected in a Linux system by first being injected into a process in an allow-list. Therefore, fileless malware poses a threat to Linux-powered IoT devices and infrastructure. As a result, creating a method to defend a Linux system against fileless malware is crucial.

Fileless malware is one of the stealth methods that malware developers have recently used most frequently. Even though fileless malware is not a new attack vector, its high success rate has increased its use [1]. A 2017 study found that fileless virus attack strategies were used in 77% of practical assault situations [2]. The study demonstrates that conventional antivirus software cannot safeguard computer systems completely. Additionally, according to a report [3] published in 2020, the rate of fileless malware rose by 900% from 2019 to 2020. Nowadays, many crypto miners and ransomware transmit their infections to hosts via fileless malware.

Fileless malware is stored in the address space of a lawful procedure. Traditional antivirus software, on the other hand, recognizes malware based on the harmful code included in a file. Signature-based antivirus cannot access the destructive code kept in physical memory, so it cannot identify associated threats. Windows hosts are initially the most popular targets of fileless malware. The popularity of Linux-based desktops, IoT devices, and servers has recently been accompanied by the emergence of fileless malware on Linux platforms [4], and this trend is continuing. When building fileless malware for Windows computers, different techniques are used than when doing so for Linux systems [5]. The reason for this is that on the Windows and Linux platforms, the procedures and interfaces for injecting code into the memory of a process from a separate approach are dissimilar. For example, Windows fileless malware typically uses the macros of Windows Office, PDF (Portable Document Format), PowerShell, and system administration utilities like Windows Administration Interface Command (WMIC) and CertUtil. However, these have rarely been used by Linux fileless malware [6]. However, the targets of fileless malware on Linux and Windows are the same. Each aims to place malicious code within the address space of an active, legitimate process running on the target system.

Section 1 of this paper introduces the basic concepts and motivations. Section 2 provides the necessary background knowledge. Section 3 is related research that focuses on the defense mechanisms provided by the current antivirus software. Section 4 presents the system structure and implementation details of CoE, while Section 5 reports the evaluation results of CoE. The limitations of CoE are discussed in Section 6, and the final section summarizes the paper, highlighting the contributions of CoE and outlining directions for future work.

2. Background

The pertinent background information for Linux fileless malware is introduced in this section. Although the method for inserting malicious code into a process changes between Windows and Linux systems, their objectives are the same. These techniques aim to introduce malicious code into a process's address space. These techniques aim to save the code in physical memory rather than a file in long-term storage.

2.1. Fileless Malware

Malware that uses no files is both an attack method and an attack vector. The code of fileless malware is stored at the address space of a process that is now operating on a victim host, as opposed to traditional malware, whose code is stored in a file at the victim host’s permanent storage. Antivirus software must obtain the file to determine whether it is malware. It is challenging to identify fileless malware, since its code is kept in the victim host’s physical memory, and current antivirus software is not built to recover code from memory whose contents may change. As a result, an increasing number of malware developers distribute their dangerous code using fileless malware.

2.2. Approaches to Inject Code into a Process

Two main ways exist to introduce code into a Linux process’s address space. The first inserts shell code into the address space of the susceptible process by taking advantage of the buffer overflow vulnerability [7]. This injection can be completed by an internal or external cycle. This method launches heap overflow attacks [8] and stack-smashing assaults [9]. Figure 1’s Path C illustrates this path.

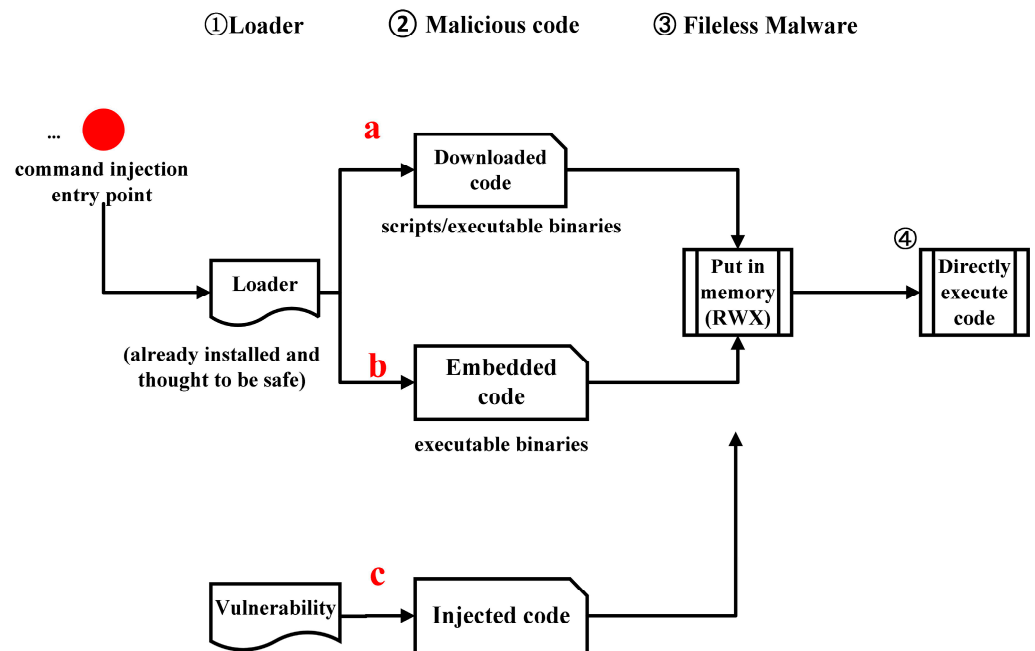


Figure 1. Possible injection and execution flows of fileless malware. a, b, and c above related arrows represent paths a, b, and c, respectively.

The target host’s internal process must cooperate with the second method [10] to accomplish code injection. Attacks using command injection that are launched from outside the target host [11] are used to carry out this operation. This procedure, typically a Linux shell, can become Figure 1’s loader. This procedure may activate other loaders. The Python and Perl interpreters, which Linux distributions typically pre-install on a Linux system, are additional loaders. Additionally, PHP (Personal Homepage Program) is frequently used as a loader on hosting systems.

A loader can be a Linux shell process or a process the Linux shell executes. A loader is responsible for downloading harmful code from an external host to a target host. Malicious scripts or executable binaries are downloaded when a loader is a Linux shell, injecting malicious code into a process and causing the malicious code to be executed in memory. No files are modified. For example, the following shell command `$curl http://attacker/evil.pl` (accessed on 1 March 2024) | perl downloads a malicious Perl script from a remote host and invokes a Perl interpreter to execute it. All the above operations happen in memory only. In this case, the shell is the loader.

Malicious code can be either a script or an executable binary, as shown in path A of Figure 1. When an executable binary belonging to this type is executed, the related process can inject malicious code into a different process using a system called `ptrace()`. The process can also use `memfd_create()` to allocate a memory area in the address space. Then, the process injects a malicious ELF file into the allocated memory area. Finally, the process executes the malicious ELF directly, without touching any other file in the permanent storage of the target host. If the downloaded code is malicious, the script can use a system called `memfd_create()` to perform the same thing described above. The malicious code shown in path b of Figure 1 is also an executable binary, but its data area already contains malicious machine code. By transferring execution flow to the malicious machine code in the data area, a process belonging to this type can execute the malicious machine code directly. All related files are deleted from their storage after creating any of the above process types, and malicious code is executed. Using the `ptrace()` system, an injecting process can inject code into a different process (injected process). However, to do so, the injecting process must first collect the PID (Process IDentifier) and memory layout of the injected process on the target host. As a result, it is often more practical, easier, and less detectable to create fileless malware at the target host using the `memfd_create()` system call.

2.3. An Execution Example of Fileless Malware

This section presents an example of how fileless malware can be spread to a host and executed. Figure 2 illustrates this example. The target host in this figure has a command injection vulnerability, which allows an attacker to perform a shell on the target host by injecting malicious commands into a benign command. If the target host has Perl or Python interpreters, commonly pre-installed on many operating systems, the attacker can use these interpreters to execute malicious code. In that case, an attacker can use the shell to execute the following commands: `$curl http://attacker/evil.pl` (accessed on 1 March 2024) `| perl` or `$curl http://attacker/evil.py` (accessed on 1 March 2024) `| python`. Either command downloads a script from a remote host and then invokes an interpreter to execute the script. The above operations do not create a file in the permanent storage of the target host.



Figure 2. Fileless malware attack example.

In the Perl script example, the Perl interpreter creates a process to execute the downloaded script, `evil_elf.pl`. It first uses the `memfd_create()` system call to set up an anonymous file in the process's address space, then writes a malicious ELF binary into this file, and finally executes the malicious binary. The malicious ELF still resides in the memory of the process, not in a file at the permanent storage of the target host. As a result, the download operation, injection operation, and execution operation occur in the target host's memory. Since the `memfd_create()` system call first appeared in Linux version 3.17 in October 2014, this example is applicable only to Linux distributions with Kernel version 3.17 or newer.

The function of `memfd_create()` is similar to `malloc()`. The difference is that `malloc()` returns an indicator that points to an allocated memory. `memfd_create()` creates an anonymous file in the memory and returns its file descriptor. `memfd_create()` was initially designed to allow various programs to share a memory and exchange messages through file descriptors. This kind of file is similar to a regular file function, with the write and read

functions, and can also be loaded into memory for execution. However, only the link file can be seen in the file system. This anonymous file does not exist on a physical hard disk. A link file is a particular file that points to another file.

3. Related Work

This section introduces the current defense methods and characteristics of antivirus software, analyzes the defense effect against fileless attacks, and points out their shortcomings. Antivirus software that can detect fileless malware is typically designed for Windows. Finding such antivirus software for Linux systems is rare, if not impossible.

3.1. Fileless Malware Collections, Creation, and Analyses

Antivirus software that detects fileless malware is usually developed for Windows in the industry. Finding such antivirus software in a Linux system is rare, if not impossible. Hence, the major effort for Linux fileless malware is to analyze the behaviors of Linux fileless malware. Fan Dang et al. [12] deployed hardware and software IoT honeypots to collect fileless attacks on Linux-based IoT devices in the wild. They also analyzed the collected samples for their prevalence, exploits, environments, and impacts. B.N. Sanjay et al. [13] conducted a detailed survey on fileless malware, especially Windows fileless malware. Sherif Saad et al. [14] designed and implemented a fileless malware using new features in JavaScript and HTML5. They tested their proposed fileless malware with several free and commercial malware detection tools that apply static and dynamic analysis. Experimental results show that their fileless malware bypassed all the anti-malware detection tools used in their study. In a 2021 malware survey about diverse notorious malware made by [15], Caviglione et al. concluded, "... standard mechanisms such as system monitoring, firewalling, and proxying, restricted access to command prompts, website analysis, whitelisting, and user education could be ineffective. Thus, research is needed to detect and counteract fileless threats efficiently".

3.2. Static Analysis

One of the static analysis techniques antivirus software uses is signature-based technology, which most antivirus software adopts. As shown in Figure 3, this type of antivirus software scans a file to generate its signature. Then, it looks up the virus signature database to see whether the file's signature matches the virus's signature in its virus signature database. The file is deemed a virus file if there is a match. Otherwise, it is supposed to be a normal file. Antivirus software destroys the virus file and issues a warning message to the computer users. Each antivirus software company maintains its virus signature database. The quality of the virus signature database determines the detection precision of an antivirus. Because virus signature databases are the signatures of zero-day viruses, static analysis-based antivirus cannot detect zero-day viruses. Because new viruses continue emerging, antivirus software companies must collect new virus signatures persistently and update their virus signature database continuously to mitigate the influence time of zero-day viruses.

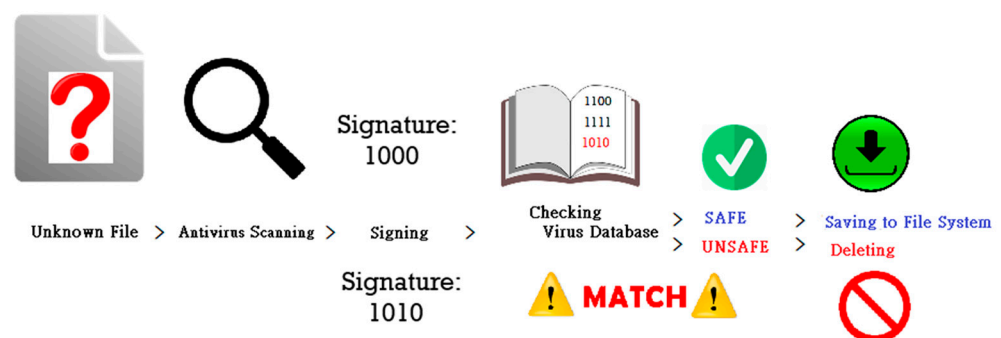


Figure 3. Execution steps of static analyses.

This time-consuming job requires experienced professionals to maintain the virus signature database. Besides, to avoid being detected by antivirus software, virus makers develop various approaches to change the forms of their viruses. Encryption and compression are the most used transformation methods [16]. An antivirus must obtain the file before applying static analysis to check whether a file is a virus file. However, the fileless malware's code resides in a host's memory, not the host's file. Thus, it is challenging for static analysis-based antivirus software to detect fileless malware. After all, current static analysis-based antivirus software cannot retrieve malicious code stored in memory.

3.3. Dynamic Analysis

Dynamic analysis is the other virus detection method. Researchers have found that most viruses have some special behavior patterns. These behavior patterns are relatively uncommon in normal programs. Therefore, dynamic analysis-based antivirus software uses these patterns to identify whether a running process is a virus. For example, normal programs call graphics APIs to draw interfaces first, but viruses usually start reading and writing a hard drive directly and download other malicious programs. Dynamic analysis-based antivirus can detect the existence of fileless malware. Dynamic analysis-based antivirus software usually creates an isolated virtual environment first and then executes the program it wants to check in this virtual environment.

Meanwhile, when the program is executed in an isolated environment, dynamic analysis-based antivirus collects behaviors generated by the program. Finally, based on the behaviors that the antivirus software contains, the antivirus software determines whether the program is a virus. For example, McAfee utilizes behavior and dynamic analysis to detect whether a program runs simultaneously with PowerShell.

Compared with static analysis-based virus detection approaches, dynamic analysis-based virus detection approaches usually use more resources and may create more false alarms. Besides, it is not difficult for malware makers to change the behaviors of their malware while completing the same work. Malware makers may add some operations to bypass detection. For instance, a virus can elude detection by introducing a delay of 100 to 200 milliseconds to a harmful command. Moreover, some malware will first detect whether it is in an isolated virtual environment, such as a virtual machine or sandbox. It does not execute malicious code if it finds it is in a remote virtual environment.

3.4. Security Settings to Block Fileless Malware Execution

Security settings can turn off some functions that may be abused by fileless malware to dispatch it to various hosts. Usually, they are radical solutions and may influence users' usage experience of some programs. This defense method is not recommended if a user needs a more open environment. Two examples utilize security settings to protect a system against fileless malware. First, Microsoft Office users can turn off the macros to avoid fileless malware that is spread through macros. Second, Web browser users can turn off the JavaScript execution capability of their browsers to prevent related attacks. However, this may make most websites work abnormally.

Providing users with setting security policies for a Linux system can achieve access restrictions, which can be set to prohibit the execution of fileless malware. Because fileless malware may utilize the directories used by shared memory, such as /tmp or /dev/sh/, some administrators may mark these directories as non-executable to protect their systems from fileless malware that uses these directories. But a super user can still execute the programs in these directories.

3.5. Adoption of Security Patterns

Recent studies have underscored the critical role that security patterns play in enhancing the resilience of security frameworks like Check-on-Execution (CoE). In 2016, Hamid et al. [17] researched establishing a formalized approach to modeling security patterns, which are reusable solutions to common security problems in software design. By formal-

izing these patterns, the authors aimed to provide a structured methodology that can be systematically applied during the software development lifecycle to enhance the security of software systems. They integrated security concerns early in software development, mainly through pattern-based approaches. Security patterns encapsulate best practices for addressing specific security requirements, allowing developers to implement security measures consistently and effectively. The research explored the challenges associated with the informal nature of existing security patterns, often leading to ambiguities and inconsistencies in their application. To address these challenges, the authors proposed a formalized framework for modeling security patterns, which includes using formal languages and tools to ensure that security patterns are precisely defined, analyzed, and implemented.

In 2022, Fernandez et al. [18] explored how ASPs can create a systematic, reusable framework for addressing software and system design security challenges. Abstract Security Patterns are generalized, high-level templates that describe security mechanisms without tying them to specific implementations or technologies. These patterns encapsulate fundamental security principles that can be applied across various domains, providing a flexible yet robust foundation for securing software systems. The authors argued that by using ASPs, developers and architects can ensure that security considerations are inherently built into the system's architecture from the early design stages.

4. Goals, Principles, and System Structure

This section describes the goals, principles, system structure, and significant components of CoE. Unlike the approaches adopted by system settings (Section 3.4), which may turn off some system functions, CoE does not turn off any system-provided tasks while protecting the system from fileless malware. Additionally, the impact of CoE on the performance of normal processes is minimal. A typical process is defined as one that does not execute code stored in a writable memory area. For legitimate processes that need to execute code stored in a writable area, such as a heap, if the system maintains the hash value of the code confirmed to be benign, then only the performance of the first execution of these processes will be affected. Subsequent executions will not experience any significant performance degradation.

4.1. Design Principles

CoE is designed based on the following principles. For security reasons, a writable page of a process is usually not executable. However, for specific purposes, such as in a heap area, a writable region may still need to be both writable and executable simultaneously. Writable process areas are often targets for fileless malware injection. Therefore, if we pause the execution of code stored in a writable region, collect the code, and determine whether it is malicious, we can effectively identify and stop harmful code injected into a process. It enables us to protect a system from fileless malware. At the same time, legitimate processes can still use writable memory to store and execute their code, provided that we allow the suspended process to continue running when the analysis shows that the code is not a virus.

4.2. System Structure

Figure 4 shows the system structure of CoE. CoE consists of the following major components: CoE System Call Interceptor, textitCoE Page Fault Handler, CoE Scanner, CoE File Extractor, CoE Code Extractor, and CoE Packer, except for component CoE Scanner, which is in a user space process. The Linux kernel address space contains all the remaining components. The primary element within the CoE System Call Interceptor is the CoE File Extractor. Meanwhile, the CoE Page Fault Handler is constructed upon the foundations of the Linux Page Fault Handler, but with the addition of two extra components: the CoE Code Extractor and CoE Packer. As mentioned in Section 2.2, there are different approaches for fileless malware to inject code into the address space of a process. We use component

CoE System Call Interceptor and component CoE Page Fault Handler to intercept and catch the code stored in a process’s memory.

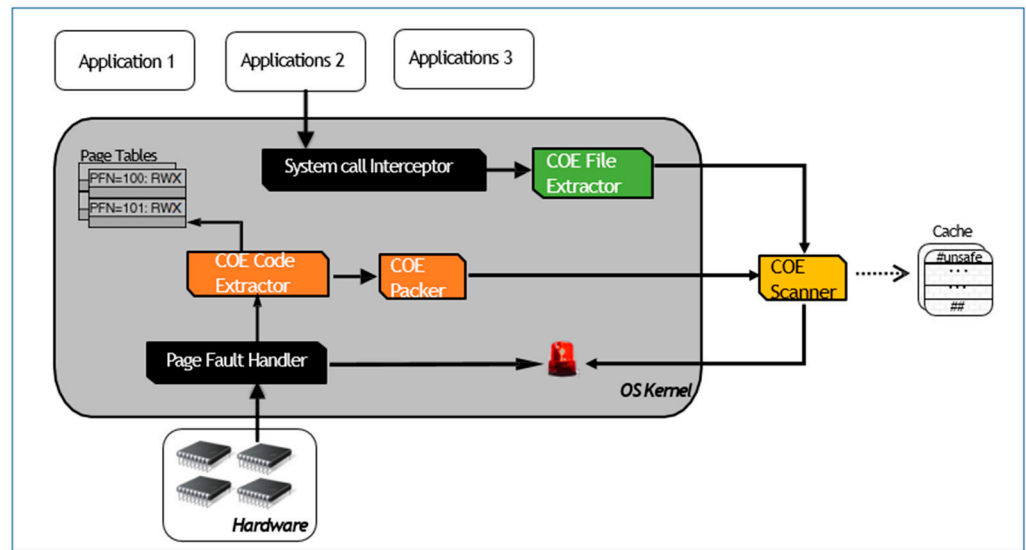


Figure 4. System structure of CoE. (where # is used to denote comments).

1. **CoE Code Extractor:** The primary function of the CoE Code Extractor is to extract code from memory. CoE Code Extractor utilizes the NX bit (No eXecute bit) provided by AMD (Advanced Micro Device) 64-bit CPUs (Central Processing Unit) or the XD bit (Execute Disable bit) provided by Intel 64-bit CPUs, both of which are supported by Linux, to trigger its execution. Figure 5 shows the layout of a Page Table Entry (PTE) and the position of the NX bit in a PTE. When the NX bit of a PTE for a page frame is set to 1, it indicates that the page frame is not executable. A Page Fault Exception will be triggered if the CPU attempts to execute code from this page frame. Currently, the CoE Code Extractor is executed within the CoE Page Fault Handler. CoE preemptively sets the NX bits of all PTEs corresponding to writable and executable page frames to 1. This approach allows CoE to leverage hardware to detect when the CPU attempts to execute code stored in a writable page frame, eliminating the need for time-consuming software checks.

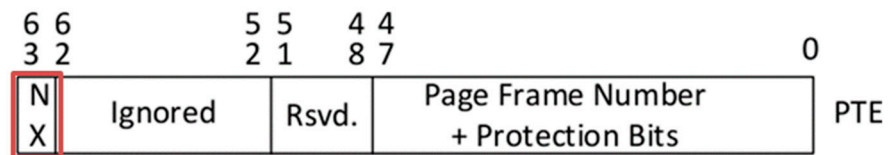


Figure 5. NX bit of a PTE(The numbers are likely from a 64-bit system, as indicated by the range 63-0.).

The copy-on-write mechanism, adopted by most modern OSes, including Linux, inspired the design of the execution triggering mechanism of the CoE Code Extractor. Figures 6 and 7 illustrate how copy-on-write works. As shown in Figure 6, there are two tasks (i.e., processes), task A and task B, sharing a physical page frame. The PTE in task A for the page frame, PTEA, is marked as non-writable. Similarly, the PTE in task B for the same page frame is also marked as non-writable. When task A attempts to write data into this page frame, a Page Fault Exception occurs because the PTE is set to non-writable. Suppose the Page Fault Handler discovers that the virtual memory area assigned to task A designates the page frame as writable. In that case, it recognizes that the copy-on-write mechanism triggered the fault. Consequently, the handler allocates a new page frame to

task A, updates PTEA to point to this new page frame, and sets the new page frame as writable in PTEA. Figure 7 shows the result.

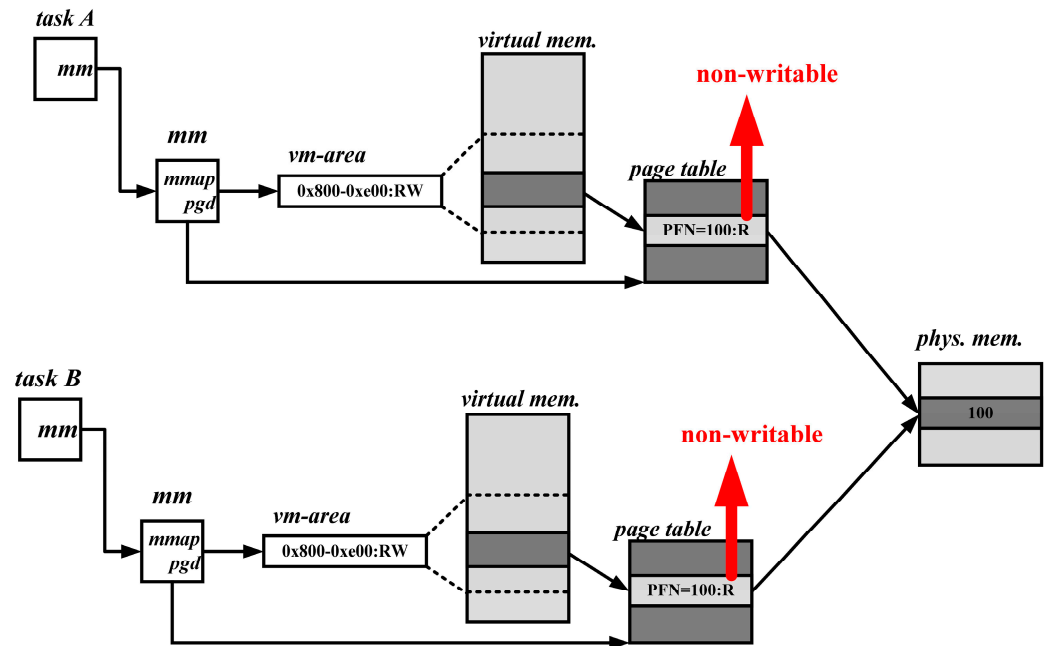


Figure 6. Before copy-on-write.

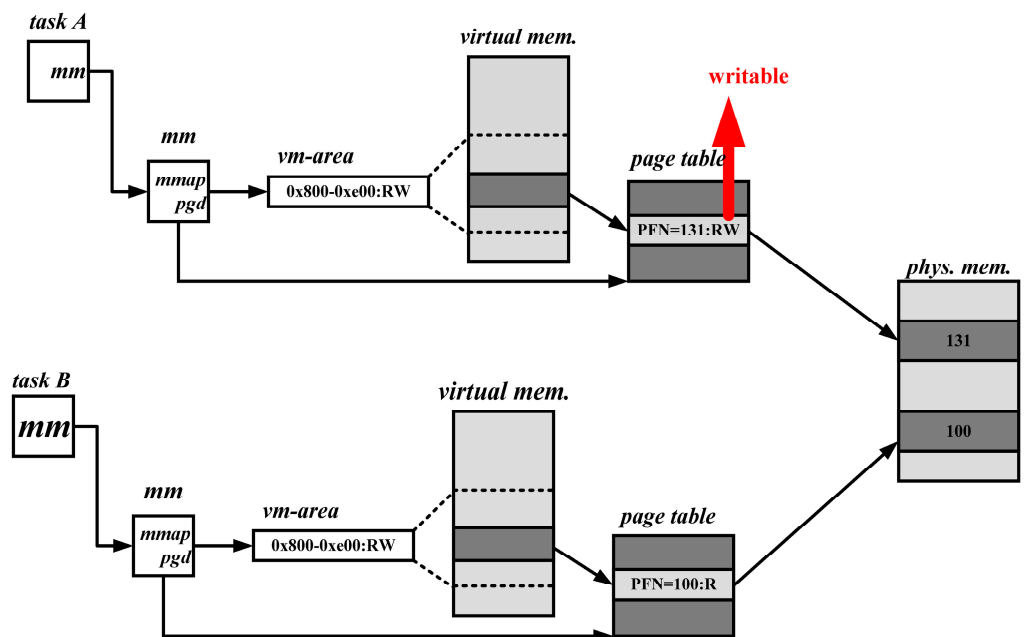


Figure 7. After copy-on-write.

CoE uses the NX bit and a mechanism similar to copy-on-write to trigger the execution of the CoE Code Extractor. In Figure 8, the page frame with the number 100 in the rightmost square represents a page frame that is readable, writable, and executable. Task A's corresponding *vm-area* will retain a record of these permissions. Because the page frame is writable and executable, CoE first sets the PTE for this page frame as non-executable. As a result, when task A attempts to execute code stored in this page frame, a page fault is issued, triggering the CoE Code Extractor. CoE then checks the executable permission of the page frame in the related *vm-area*. If the page frame is not executable, the CoE Code Extractor terminates the execution of task A. If the page frame is executable, CoE collects

the code in the page frame and sends it to other CoE components for analysis. If the code is found to be malicious, CoE terminates task A. Otherwise, CoE sets the page frame as executable in its PTE and allows task A to resume execution, as shown in Figure 9.

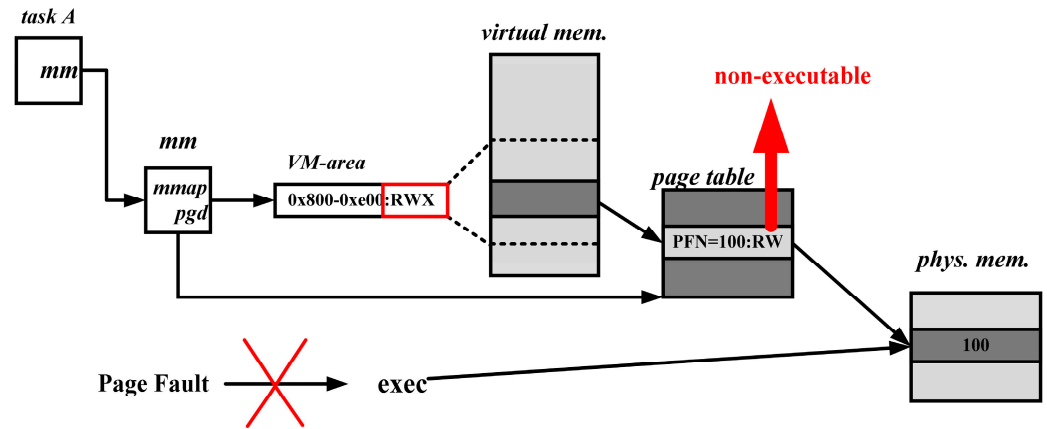


Figure 8. Before the execution of CoE Code Extractor. (where RWX stands for Read, Write, and Execute, with “X” indicating that execution is not allowed).

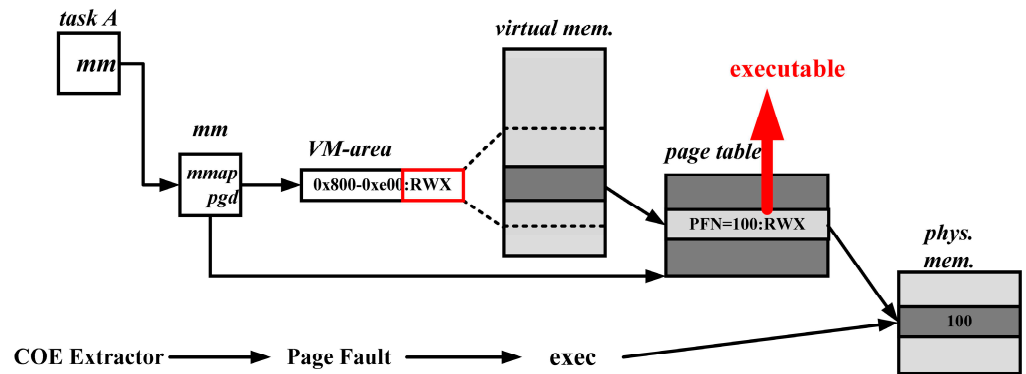


Figure 9. After the execution of CoE Code Extractor. where RWX stands for Read, Write, and Execute.

2. CoE Packer: The primary function of the CoE Packer is to convert the code extracted from memory into ELF format. Once the CoE Code Extractor determines the necessity of verifying a code segment stored in a writable region, it instructs the CoE Packer to retrieve the code and package it into an ELF file. The CoE Packer accomplishes this by appending the appropriate ELF header and setting the ELF header’s entry point field to the starting point of the assembled code. This conversion allows CoE to transfer the code into an ELF file, which VirusTotal can then check to determine whether the code is malicious or benign. VirusTotal only accepts and analyzes data in file form.
3. CoE Scanner: The primary function of the CoE Scanner is to scan packed ELF files. After the CoE Code Packer converts a piece of code into an ELF file, it hands it over to the CoE Scanner to determine whether the code is malware. The current version of CoE Scanner utilizes VirusTotal’s APIs to assist in identifying malicious code. VirusTotal provides a malware analysis service leveraging over 60 antivirus engines to analyze input files, offering free and paid options.
4. CoE File Checker: As described in Section 2.2, file malware may use the `memfd_create()` system call to create an anonymous area to inject a malware file and then use the `execve()` system call to execute the file hidden inside the address space of a process. CoE File Checker intercepts and collects files executed in this manner. When a program uses the `execve()` system call, the CoE File Checker checks the file’s location to be executed and determines whether the file is in memory. If the file is in memory, the CoE File Checker

pauses the execution, retrieves it, and submits it to the CoE Scanner for further analysis. CoE File Checker only allows benign programs to continue their execution.

4.3. Execution Paths of CoE

CoE uses two execution paths, the file path and the text path, to handle fileless malware. The file path consists of the CoE File Extractor and CoE Scanner components, which handle fileless malware created by the operating system called `memfd_create()` and `exec()`. As described in subsection II-B, fileless malware created this way is more common, accessible, and stealthy. Unlike fileless malware developed by the `ptrace()` system call, this type does not require collecting information about an injected process, such as PID and memory layout, from the injecting process at the target host. The text path consists of the CoE Code Extractor, CoE Packer, and CoE Scanner components, which handle fileless malware that contains only machine code in an attack process. This machine code may be stored in a writable and executable memory area of the process or injected into the process using a system called `sptrace()`.

1. Execution Flows of the File Path: When CoE detects a process using the `execve()` system call to execute a program, the CoE File Extractor first checks whether the file is in memory, as illustrated in Figure 10. If the file is not stored in memory, the CoE File Extractor allows the process to execute it. However, if the file is stored in memory, the CoE File Extractor retrieves it and passes it to the CoE Scanner. The CoE Scanner then uses VirusTotal to determine whether the file is malicious. The CoE File Extractor generally allows the system to handle the file unless it is found to be malicious, in which case the file’s execution is prevented, and the process is promptly terminated.
2. Execution Flows of the Text Path: When an instruction is executed, the CPU hardware checks the NX bit in the PTE of the corresponding page frame, as illustrated in Figure 11. If the NX bit is 0, the instruction is allowed to execute. However, if the NX bit is 1, the hardware triggers a Page Fault Exception. This exception verifies whether the permission in the vm-area of the page frame is writable. If it is writable, execution is transferred to the CoE Code Extractor to determine whether the consent in the vm-area is marked as executable. The Page Fault Handler terminates the related process if the consent is marked as non-executable. If the permission is marked as executable, the CoE Code Extractor restores the NX bit to 0. Subsequently, the CoE Packer retrieves the code from memory and packs it with an ELF header, transforming it into an ELF file. The ELF file is then sent to the CoE Scanner, which scans the file using VirusTotal to determine whether the code is malicious or benign. If the file is harmless, the CoE Scanner resumes the execution of the code; otherwise, it terminates the execution. The size of a page frame is 4 K.

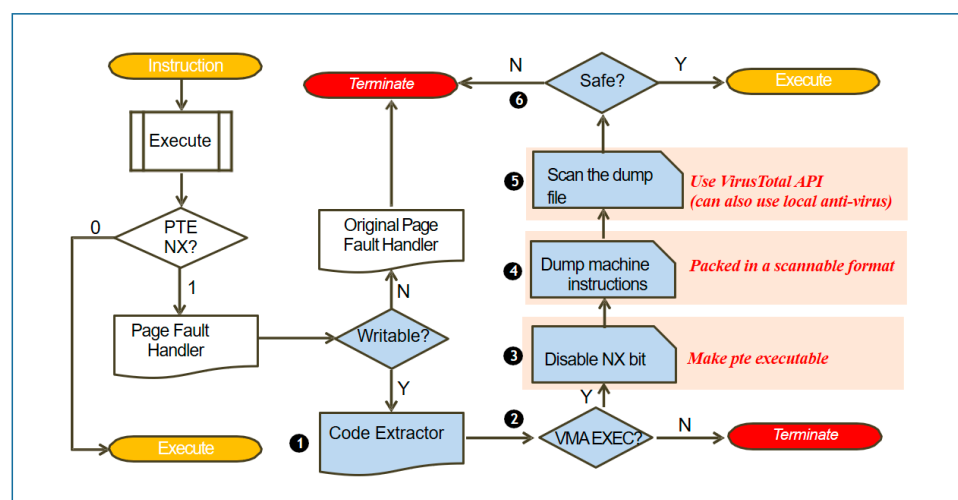


Figure 10. Flowchart of the file path. (The numbers represent the steps of execution.)

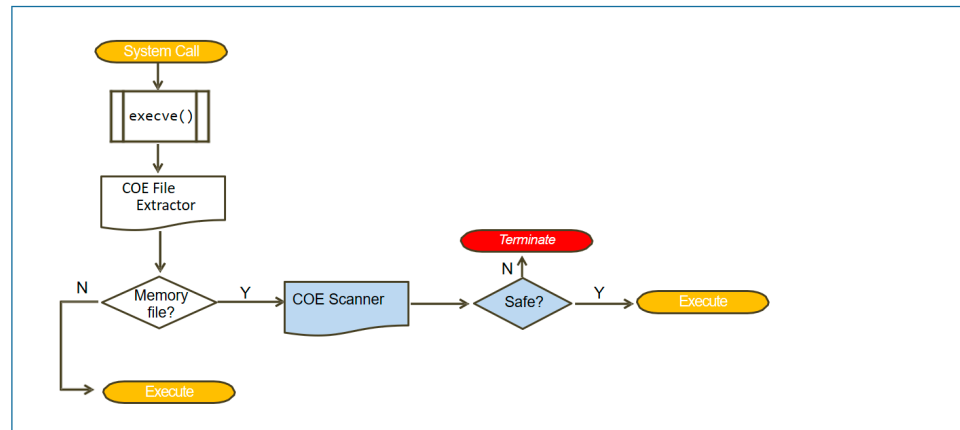


Figure 11. Flowchart of the text path.

5. Evaluation and Analysis

This section presents the results of various experiments conducted to evaluate the effectiveness and efficiency of CoE. The analysis of these experimental results aims to determine whether CoE can protect a Linux system from attacks by various fileless malware.

5.1. Effectiveness Tests for Code Stored in the Stack Segment

The host, OS, and Linux kernel specifications used in our experiments are detailed in Table 1. To determine if CoE can pause the execution of code stored in a writable memory area, we created two programs: `safe_code.c` (displayed in Table 2) and `unsafe_code.c` (displayed in Table 3). These programs contain machine code stored in a local array variable, with execution transferred to the stored machine code. To test this ability, we retrieved the code, packed it into an ELF file, and sent it to VirusTotal for analysis. In `safe_code.c`, the machine code modifies the value of a local variable `a` to 10. In `unsafe_code.c`, the machine code is a shell code commonly used in buffer overflow attacks. We allowed the test process stack to be executable during the execution of these programs, demonstrating the thoroughness of our analysis.

Table 1. CPU, OS, and Linux kernel specifications.

CPU	3.4 GHZ AMD Ryzen 7 1700X CPU with 8 cores
OS	Ubuntu 18.04
Kernel	Linux Kernel 4.20.3

Table 2. Content of file `safe_code.c`.

```

1  #include <stdio.h>
2  #include <unistd.h>
3  int main() {
4      printf("pid:%d\n", getpid());
5      int a = 0;
6      char add[] = {
7          "\x55",          // push  %rbp
8          "\x48\x89\xe5",  // mov  %rsp,%rbp
9          "\x48\x89\x7d\xf8", // mov  %rdi,-0x8(%rbp)
10         "\x48\x8b\x45\xf8", // mov  -0x8(%rbp),%rax
11         "\xc7\x00\x0a\x00\x00\x00", // movl  $0xa, (%rax)
12         "\x90",          // nop
13         "\x5d",          // pop  %rbp
14         "\xc3",          // retq
15     };
16     printf("%d\n", a);
17     printf("%p\n", add);
18     (*(void (*)())add>(&a); // a=10;
19     printf("%d\n", a);
20 }
    
```

Table 3. Content of file unsafe_code.c.

```

1  #include <stdio.h>
2  #include <unistd.h>
3  int main() {
4      printf("pid:%d\n", getpid());
5      char shellcode[] = {
6          "\x48\x31\xd2", // xor   rdx, rdx
7          "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68", // mov
8          $0x68732f6e69622f2f, %rbx
9          "\x48\xc1\xeb\x08", // shr   $0x8, %rbx
10         "\x53" //push  %ebx
11         "\x48\x89\xe7" // mov  %rsp, %rdi
12         "\x50" //push  %rax
13         "\x57" //push  %rdi
14         "\x48\x89\xe6" //mov  %rsp, %rsi
15         "\xb0\x3b" //mov  $0x3b, %al
16         "\x0f\x05" // syscall
17     };
18     printf("%p\n", shellcode);
19     (*(void (*)())shellcode)();
20 }

```

The program `safe_code.c` is executed successfully and subsequently terminates its execution. Its pattern is similar to that of a traditional buffer overflow attack. Experimental results indicate that CoE can protect a system from fileless malware, including buffer and heap overflow attacks. However, this paper focuses solely on fileless malware. In addition to the previous experiments, we conducted further tests to assess the current efficacy of antivirus engines against fileless malware. Clam AntiVirus (ClamAV) is a popular open-source Linux antivirus engine. We initially selected some viruses that ClamAV could detect for these tests. We then converted these viruses into fileless malware and provided the newly created fileless malware to ClamAV for examination. ClamAV failed to identify the fileless malware as containing viruses. These results demonstrate that detecting fileless malware remains challenging for antivirus engines, corroborating the conclusions of Caviglione et al. regarding fileless malware, as discussed in Section 3.1.

5.2. Effectiveness Evaluation

To evaluate the effectiveness of CoE, we conducted experiments using three different types of malware obtained from various sources. The first type was sourced from VirusShare, the second consisted of shell code collected from the Shell-Storm Database, and the third was packed malware. For our experiments, we created packed malware by selecting samples from the first type of malware. We extracted each selected piece's text segment and added our ELF header and entry point to form a new file. We then used a packing tool to pack the new file. During the execution of these three types of malware, whether directly or indirectly through a process, CoE was able to identify and halt their execution. Additionally, CoE generated or retrieved the relevant ELF files for scanning. The detection effectiveness of CoE depended entirely on the virus signature databases we utilized. For our experiments, we relied on VirusTotal to determine if a file was malicious for our experiments.

Case 1: In this experiment, we selected 2668 malware samples from VirusShare, converted them into fileless malware, and executed them. CoE's file path was triggered to handle the fileless malware. Among the 2668 malware samples, CoE correctly identified 2661 as malware and mistakenly classified seven fileless malware samples as benign. The detection rate was 99.73%. Of the seven malware samples misclassified as harmless, five could not execute, one merely printed a message, and the last attempted to implement a program in the `tmp` directory. Since none of these seven samples exhibited malicious behaviors, they should not be considered malware.

[Result analysis]: Experimental results show that CoE can effectively detect fileless malware.

Case 2: We collected 15 pieces of Linux x86-64 shell code from the Shell-Storm Database to create 15 pieces of fileless malware. CoE's text path was triggered to handle these fileless malware samples. CoE successfully suspended the execution of all 15 pieces and obtained the related ELF files. However, only eight out of the fifteen pieces of fileless malware were detected as viruses by VirusTotal, resulting in a detection rate of 53.33%. Table 4 details the types of shell code samples used in our experiments and the detection rates for each type.

Table 4. Detection Rates of Fileless Malware Created by Shell Code.

Shell Code Form	# of Detected Shell Code Samples	# of Total Shell Code Samples	Detection Rate
Execute ("/bin/sh")	2	4	50%
Connect Back Shell	3	7	42.85%
Execute ("shutdown -h now")	1	2	50%
Add users with passwd	1	1	100%
Execute ("/sbin/pagetables -F")	1	1	100%
Total	8	15	53.33%

[Result analysis]: As mentioned above, CoE successfully suspended the execution of all 15 pieces of fileless malware and obtained the related ELF files, demonstrating that all CoE components functioned correctly. The CoE can protect a system against shell code attacks despite being primarily designed to defend against fileless malware. If the virus signature databases used by CoE can incorporate more shell code signatures, or if we create a dedicated CoE shell code signature database, the accuracy of CoE's shell code detection could be significantly improved.

Case 3: Packing is a technique that compresses or encrypts malware samples to alter their signatures, allowing attackers to create mutations that bypass antivirus detection. When packed malware is executed, the decompression or decryption code is first applied to the compressed/encrypted code, which is then stored in a writable and executable area. This process triggers CoE's execution. Our experiments involved 112 pieces of packed malware. CoE detected 70 out of the 112 samples, resulting in a detection rate of 62.5%.

[Result analysis]: The malware used in case 3's experiments is the same as in case 1's. In case 1, experiments demonstrated that CoE could detect almost all tested fileless malware. However, in case 3, the detection rate of CoE dropped to only 62.5%. The reason for this discrepancy is as follows: For a file called file test retrieved from VirusShare, case 1 used it directly, allowing CoE to send the complete file test to VirusTotal. In contrast, case 3 involved altering the ELF header of the test. Consequently, even though the packed file received by VirusTotal contained the same code segment as the file test, its different ELF header affected the detection process.

These results suggest that some antivirus engines consider both the ELF header and the code to determine whether a file is malware. Thus, as concluded from the analysis of case 2's results (shell code case), incorporating signatures derived from malicious machine code can enhance CoE's detection rate for packed malware.

5.3. Efficiency Evaluation

We conducted experiments to measure the execution overhead introduced by CoE. Since CoE affects the execution time of code stored in writable and executable areas and memory areas created by the `memfd_create()` system call, we evaluated the performance overhead in these two scenarios.

Figure 12 shows that the performance overhead for executing code stored in a writable and executable area is 22%. However, a program's code section is typically non-writable for security and reliability reasons, meaning this overhead should not impact most programs.

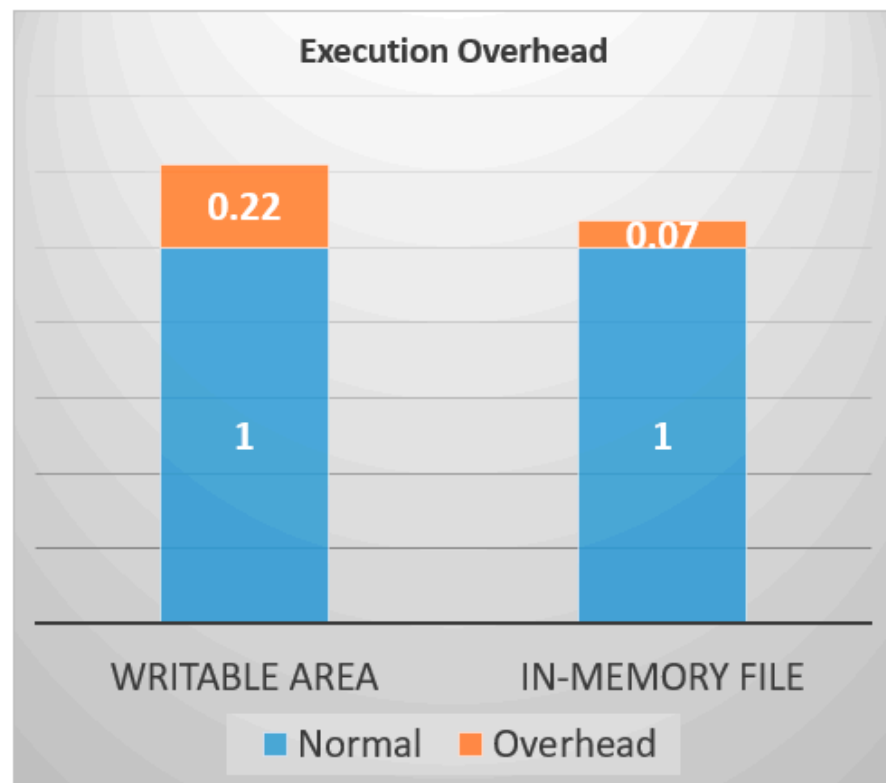


Figure 12. Execution performance overhead introduced by CoE.

Figure 12 indicates that the performance overhead for executing a file stored in a memory area created by the `memfd_create()` system call is 7%, considered trivial.

5.4. File Scanning Time Evaluation

The primary performance overhead introduced by CoE, the file scanning time, includes the time VirusTotal needs to analyze a file and determine its nature and the time to transmit a file between the host and VirusTotal. VirusTotal returns the existing scan results when it receives a previously analyzed file. However, when it encounters a new file, it performs an initial analysis before sending back the results, which takes longer. It is important to note that this latter scenario is relatively rare, providing reassurance about the frequency of the issue.

In our first experiment, we repeatedly sent the same file that VirusTotal had already analyzed and calculated the average file scanning time. As shown in Figure 13, the average scanning time was 4.71 s. In the second experiment, we sent files that VirusTotal had not previously analyzed and calculated the average scanning time. Figure 14 shows that the average scanning time in this case was 80.1 s. These results indicate that file scanning time represents a significant overhead.

To address this issue, caching previous file scanning results can minimize the need for repeated scans of the same file. Additionally, triggering the scan of related files during program installation can mitigate this non-trivial file scanning time issue. Furthermore, using a local antivirus engine for scanning instead of sending files to VirusTotal may also reduce the file scanning overhead.

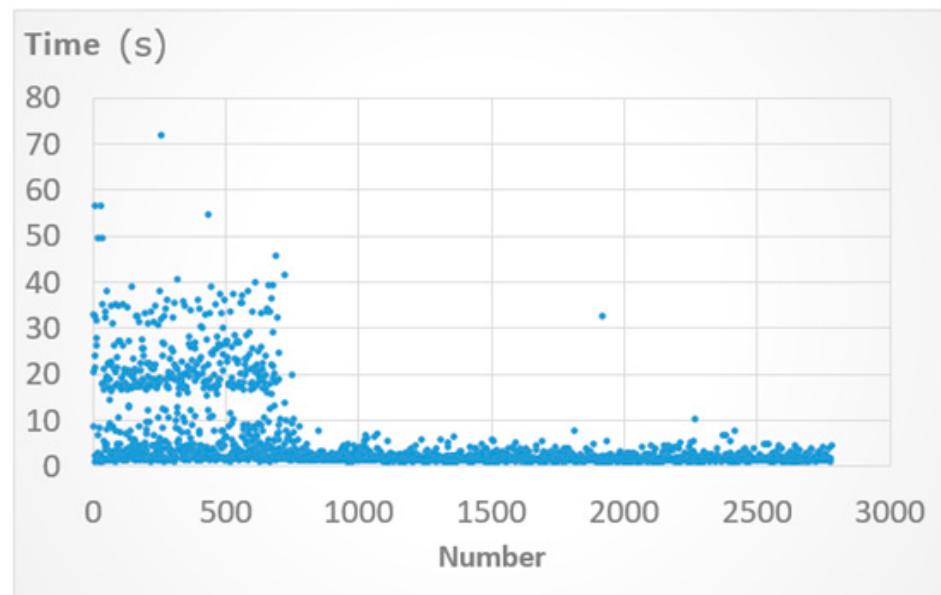


Figure 13. Average file scanning time for a file that VirusTotal has analyzed.

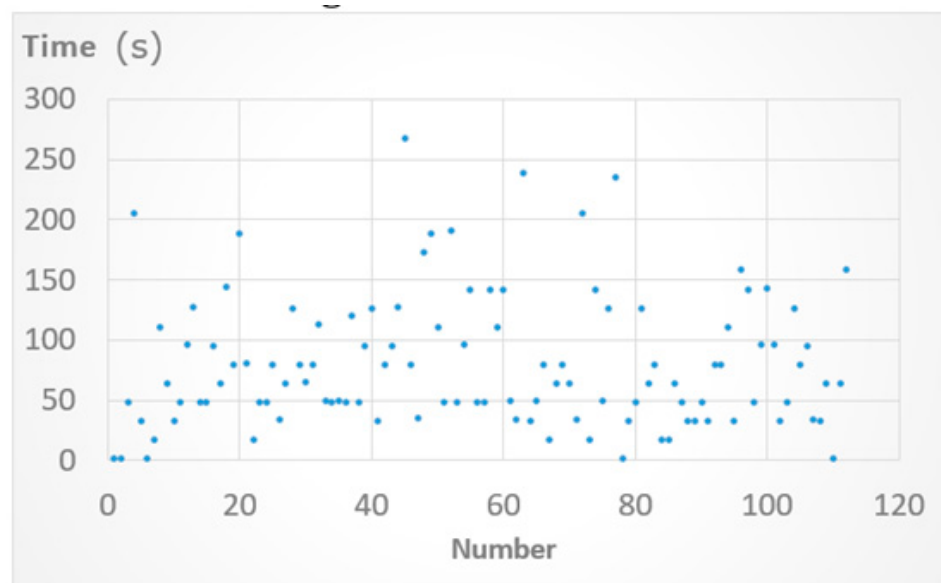


Figure 14. Average file scanning time for files that were not analyzed by VirusTotal before.

6. Discussion

CoE utilizes CPU hardware to initiate execution when code is about to be run from a memory area that is both writable and executable. This approach is logically sound, as it would be challenging for an attacker to inject code into a memory area that is not writable. Typically, this assumption holds. However, if an attacker employs the `ptrace()` system call for code injection, it becomes possible to inject code into a non-writable but executable memory area. This is because `ptrace()` allows code injection within the kernel address space, enabling the kernel to write code into any part of a process's address space.

We propose hooking the `ptrace()` system call to address this vulnerability. By doing so, whenever `ptrace()` attempts to write into a non-writable but executable memory area, we can dynamically alter the permissions of the related virtual memory area to be writable and executable. This measure effectively reduces the risk associated with `ptrace()`-based code injections.

7. Future Work

7.1. Integration of the Check-on-Execution (CoE) Framework into TPM Technology

To integrate the Check-on-Execution (CoE) framework into Trusted Platform Module (TPM) technology, we can significantly enhance the security of Linux systems by leveraging TPM's advanced capabilities in secure boot processes, hardware-based attestation, and integrity verification. Combining CoE and TPM creates a robust security framework that addresses various vulnerabilities in Linux systems, especially those targeted by advanced threats such as fileless malware. This integration not only provides hardware-based assurances for code execution but also ensures the overall integrity of the system from boot to runtime. Below is a detailed explanation of how CoE could be integrated with TPM, based on the concepts outlined.

The Trusted Platform Module (TPM) is a specialized hardware-based security device embedded in computing devices, offering a range of cryptographic functions and secure storage capabilities. It acts as a trusted anchor for various security-related operations. One of its primary functions is ensuring a safe boot, where TPM validates the integrity of the boot loader and kernel to ensure the system boots with only trusted software. This involves measuring and verifying each stage of the boot process to confirm that it has not been tampered with. TPM also plays a crucial role in attestation by providing proof that a system's hardware and software configurations have not been altered. Through remote attestation, TPM measures and reports the system's state to a remote verifier, ensuring the system operates securely. TPM supports sealing and binding, which involves encrypting data so it can only be decrypted when the system is in a known, trusted state. This process is essential for protecting sensitive data, ensuring it remains inaccessible if the system's integrity is compromised.

Integrating the Trusted Platform Module (TPM) with the Center of Excellence (CoE) framework involves several vital enhancements to ensure system security. The primary objective of this integration is to leverage TPM's capabilities to bolster the integrity and protection of the CoE framework at various stages. TPM enhances the integrity of the CoE framework during the boot process through secure boot integration. This is achieved by measuring and hashing the boot loader, kernel, and CoE components, with these measurements stored in the TPM's Platform Configuration Registers (PCRs). The TPM compares these measurements against known good values, allowing the system to continue booting only if the measurements match, confirming that the CoE framework is in a trusted state. Suppose any alterations or compromises are detected in the CoE framework or its dependencies, such as kernel modules. In that case, the boot process can be halted or the system can be restricted, preventing potentially malicious code from executing. TPM is used to continually verify the CoE framework's integrity after the initial boot. This is done by periodically extending TPM PCRs with the hash of the CoE's current state or the state of critical system components it interacts with, ensuring ongoing assurance of the CoE's security. Additionally, TPM's remote attestation capabilities allow the CoE framework to prove to a remote verifier, such as a security server or cloud-based management system, that it is operating in a secure and unaltered state. This feature is particularly useful in distributed environments or when higher security assurance levels are needed. TPM's sealed storage capability is utilized to protect critical CoE-related data, such as verified code signatures and configuration files. This involves encrypting and sealing critical data with TPM, ensuring it can only be decrypted when the system is in a verified good state. Furthermore, sensitive CoE operations can be bound to specific TPM states, ensuring that these operations are only executed when the system's integrity has been confirmed. To enhance security for code execution, TPM helps monitor and verify code before execution. CoE can use TPM to measure the state of memory regions or processes prior to allowing their execution, adding an extra layer of verification to ensure that only trusted code is executed. Additionally, before running any code, CoE can check if the executable's hash matches a TPM-stored whitelist of trusted executables, thereby preventing the execution of

unauthorized or malicious code. This comprehensive integration of TPM with CoE ensures robust protection and integrity throughout the system's lifecycle.

Integrating the Center of Excellence (CoE) with the Trusted Platform Module (TPM) delivers several notable benefits that significantly enhance the security of Linux systems. One of the primary advantages is increased security. By leveraging TPM, the CoE gains hardware-based security assurances, which makes it substantially more challenging for attackers to bypass or tamper with the framework. TPM's provision of a hardware root of trust offers robust protection against sophisticated threats like fileless malware, which typically exploits software vulnerabilities. TPM integration improves the overall integrity of the CoE framework and the entire system. TPM maintains the system's integrity from the boot process through runtime by continuously verifying the CoE components' integrity and system configurations' integrity. This ongoing integrity check helps prevent unauthorized modifications, mitigating the risk of security breaches and ensuring the system remains secure against potential threats. The use of TPM's attestation and sealing features ensures trustworthy code execution. TPM attestation confirms that only verified and trusted code is executed, significantly reducing the risk of fileless malware and other advanced threats that depend on executing unverified or malicious code. This integration ensures that the system only runs code verified as safe, thereby enhancing overall security and reliability.

Integrating the CoE framework with TPM technology significantly enhances the security of Linux systems by providing hardware-based assurances for code execution, system integrity, and data protection. This integration addresses potential vulnerabilities in the CoE framework, such as the risks associated with kernel-level privileges and external interactions, making it a more robust solution against advanced threats like fileless malware. By leveraging TPM's capabilities, the CoE framework can offer higher security assurance, ensuring that Linux systems remain protected in an increasingly complex threat landscape. The combination of CoE and TPM represents a powerful approach to securing modern computing environments, particularly those that rely on Linux as their operating system.

7.2. Integration of XACML into the Framework within IoT Environments

The integration of XACML (eXtensible Access Control Markup Language) into the Check-on-Execution (CoE) framework within IoT (Internet of Things) environments represents a strategic advancement in security and access control. XACML is a standard language expressing access control policies, allowing for highly granular and context-aware permission management. XACML can significantly enhance security in IoT settings, where devices are often resource-constrained and operate in dynamic, interconnected environments.

The integration of XACML (eXtensible Access Control Markup Language) into the CoE (Center of Excellence) framework offers several advantages, particularly in IoT (Internet of Things) environments. One of the primary benefits of integrating XACML into the CoE framework is its capability to enforce granular and context-aware access control policies. In traditional access control systems, permissions are often static, applied broadly to users or devices without considering the specific context in which an action is requested. XACML, however, allows for a more nuanced approach. XACML policies can be crafted to ensure that a device can only execute certain commands if it meets specific contextual criteria. These criteria could include the device's current network environment, the time of day, specific other devices in the network, or even environmental factors such as temperature or humidity. This level of detail ensures that access is granted only under appropriate and secure conditions, significantly enhancing the security of IoT devices, which often operate in varied and sometimes unpredictable environments. Integrating XACML into the CoE framework within IoT environments presents a robust solution for enhancing security through detailed and context-aware access control. This approach not only strengthens the security posture of IoT devices but also ensures that access control is adaptable to the dynamic nature of IoT networks. While challenges such as complexity and performance overhead must be addressed, the benefits of incorporating XACML outweigh these considerations. By providing a more granular and context-sensitive approach to

managing permissions, XACML significantly protects IoT devices from unauthorized code execution and other security threats, ensuring IoT systems' overall reliability and security using Linux.

8. Conclusions

This paper proposes a novel solution to combat fileless malware called Check-on-Execution (CoE) within the Linux kernel. Recent research indicates that fileless malware has successfully breached the defenses of various established antivirus engines, facilitating the spread of diverse malicious software, including notorious ransomware and cryptomining malware, across numerous hosts. Traditional antivirus engines use static analysis, maintaining a database of virus signatures. To detect malware, these engines must first acquire the file, extract its signature, and then compare it against the virus signature database. If a match is found, the file is identified as malware; otherwise, it is considered a regular program.

Once installed, fileless malware resides solely in the memory of a process, typically masquerading as a legitimate process that provides user services. This evasion strategy renders traditional static analysis antivirus engines ineffective, as they cannot even access the code of fileless malware, let alone analyze it. Additionally, the memory content of a process is constantly changing, making it crucial for a detection solution to capture the correct data—the fileless malware code—at the right time.

CoE leverages CPU hardware and Page Table Entries (PTEs) to trigger its execution, reducing processing time and increasing accuracy. As a kernel-based solution, CoE is resilient against being disabled by attackers, a common vulnerability in traditional antivirus engines. This resilience provides strong reassurance of its effectiveness. Experimental results demonstrate that CoE effectively defends systems against fileless malware. Moreover, while CoE is not explicitly designed to counter shell code attacks or packed malware, enhancing it with additional malicious code-based signatures suggests potential efficacy in addressing these significant security threats.

CoE is a kernel-based solution, similar to Address Space Layout Randomization (ASLR) and non-executable stacks, which provides robust security without requiring users to have elevated privileges. Just as ASLR offers protection to every process executing on a Linux system, CoE extends its protection to all processes running on a Linux system with CoE enabled. This intrinsic nature of CoE ensures that it operates transparently and universally, safeguarding processes without direct user interaction. Since CoE is integrated into the kernel, no external program can interact with it directly, which means that CoE does not increase the system's attack surface. This design is crucial, as it prevents CoE from being targeted or exploited by external malicious entities, thereby maintaining the system's security integrity. The misconception that CoE could increase the attack surface likely stems from its interaction with VirusTotal, managed by user-space code rather than CoE itself. This user-space code is designed to initiate outbound connections, specifically to VirusTotal, to verify potential threats. It does not accept inbound connections from any host, effectively isolating CoE from external threats and ensuring its interaction with VirusTotal does not expose the system to additional risks. However, this user-space code is flexible and can be configured to interact with alternative hosts, such as a local antivirus host like Amavis. By switching the server connection to a local host like Amavis, several benefits can be realized, including reduced network traffic, lower service costs, and faster analysis of threats. This is particularly advantageous in environments where hosts face similar attack vectors, as using a local antivirus host can significantly reduce the time required to analyze the same code sent from different local hosts. In future work, we plan to implement this capability, allowing organizations to tailor CoE's operations to their specific network environments, optimizing performance while maintaining high levels of security.

Author Contributions: M.-H.W.: visualization, writing—original draft, writing—review and editing, supervision. F.-H.H.: conceptualization, project administration, writing—review and editing. J.-H.H.: writing, investigation, and methodology. Y.-L.H.: software and methodology. H.-J.W.: software and validation. J.-X.C.: conceptualization and validation. T.-C.H.: software and validation. H.-T.Y. and K.W.: Formal analysis and Investigation.: investigation. All authors have read and agreed to the published version of the manuscript.

Funding: Taiwan’s Ministry of Science and Technology supported this work (grant number MOST 111-2221-E-008-080-MY3).

Data Availability Statement: The data supporting this study’s findings are available upon reasonable request from the corresponding author, Fu-Hau Hsu (hsufh@csie.ncu.edu.tw).

Acknowledgments: The entire study was conducted using internal resources, including the facilities and equipment provided by the authors’ institution. The authors acknowledge that no funding sources were involved in the study’s design, data collection, analysis, interpretation of results, or preparation of this manuscript. Therefore, no financial or funding-related conflicts of interest are associated with this research.

Conflicts of Interest: The authors declare that they have no conflicts of interest about the publication of this manuscript. The research presented in this study was conducted impartially and independently, without any personal, financial, or professional relationships that could be perceived as potential conflicts of interest. The authors have no affiliation with organizations or entities that may affect the objectivity or integrity of the findings. We confirm that the content of this manuscript is based solely on our work and the data collected during the study.

References

1. Alzuri, A.; Andrade, D.C.; Escobar, Y.N.; Zamora, B.M. The growth of fileless malware. 2019. Available online: <https://www.semanticscholar.org/paper/The-Growth-of-Fileless-Malware-Alzuri-Andrade/2e58298eda935452d7009ea440c838b9fc1a5658https://www.semanticscholar.org/paper/The-Growth-of-Fileless-Malware-Alzuri-Andrade/2e58298eda935452d7009ea440c838b9fc1a5658> (accessed on 5 September 2024).
2. Rayome, A.D. Report: Fileless Malware Attacks 10× More Likely to Infect Your Machine than Others. 2017. Available online: <https://www.enisa.europa.eu/publications/report-files/ETL-translations/fr/etl2020-malware-ebook-en-fr.pdf> (accessed on 5 September 2024).
3. WatchGudrd. New Research: Fileless Malware Attacks Surge by 900% and Cryptominers Make a Comeback, While Ransomware Attacks Decline. Available online: <https://www.globenewswire.com/en/newsrelease/2021/03/30/2201173/0/en/New-Research-Fileless-Malware-Attacks-Surge-by-900-and-Cryptominers-Make-a-Comeback-While-Ransomware-Attacks-Decline.html> (accessed on 5 September 2024).
4. Nick, B. Fileless Attack Detection for Linux in Preview. Available online: <https://azure.microsoft.com/zh-tw/blog/filelessattack-detection-for-linux-in-preview/> (accessed on 5 September 2024).
5. Stuart. In-Memory-Only Elf Execution (without Tmpfs). 2017. Available online: <https://magisterquis.github.io/2018/03/31/in-memory-only-elfexecution.html> (accessed on 5 September 2024).
6. Floreza, S.; Castillo, D.; Manahan, M. Security101: Defending against Fileless Malware. Available online: <https://www.trendmicro.com/vinfo/us/security/news/securitytechnology/security-101-defending-against-filelessmalware#documentexploits> (accessed on 5 September 2024).
7. Karapetyants, N.; Efanov, D. A practical approach to learning Linux vulnerabilities. *J. Comput. Virol. Hacking Tech.* **2023**, *19*, 409–418. [CrossRef]
8. Lee, Y.; Kwak, J.; Kang, J.; Jeon, Y.; Lee, B. Pspray: Timing {Side-Channel} based Linux Kernel Heap Exploitation Technique. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 6825–6842.
9. Butt, M.A.; Ajmal, Z.; Khan, Z.I.; Idrees, M.; Javed, Y. An in-depth survey of bypassing buffer overflow mitigation techniques. *Appl. Sci.* **2022**, *12*, 6702. [CrossRef]
10. CyberSecurity, F. Elf In-Memory Execution. 2018. Available online: <https://blog.fbks.ru/elf-in-memory-execution/> (accessed on 30 August 2021).
11. Sinha, S. Finding Command Injection Vulnerabilities. In *Bug Bounty Hunting for Web Security*; Apress: Berkeley, CA, USA, 2019. [CrossRef]
12. Dang, F.; Li, Z.; Liu, Y.; Zhai, E.; Chen, Q.A.; Xu, T.; Chen, Y.; Yang, J. Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud. In Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, Seoul, Republic of Korea, 17–21 June 2019.

13. Sanjay, B.N.; Rakshith, D.C.; Akash, R.B.; Hegde, D.V.V. An Approach to Detect Fileless Malware and Defend its Evasive mechanisms. In Proceedings of the 2018 3rd International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS), Bengaluru, India, 20–22 December 2018.
14. Saad, S.; Mahmood, F.; Briguglio, W.; Elmiligi, H. JSLess: A Tale of a Fileless Javascript Memory-Resident Malware. In Proceedings of the 15th International Conference, ISPEC 2019, Kuala Lumpur, Malaysia, 26–28 November 2019.
15. Caviglione, L.; Chora's, M.; Corona, I.; Janicki, A.; Mazurczyk, W.; Pawlicki, M.; Wasielewska, K. Tight Arms Race: Overview of Current Malware Threats and Trends in Their Detection. *IEEE Access* **2021**, *9*, 5371–5396. [[CrossRef](#)]
16. Sihwail, R.; Omar, K.; Ariffin, K.A.Z. Malware detection approach based on artifacts in memory image and dynamic analysis. *Appl. Sci.* **2019**, *9*, 3680. [[CrossRef](#)]
17. Hamid, B.; Gürgens, S.; Fuchs, A. Security patterns modeling and formalization for pattern-based development of secure software systems. *Innov. Syst. Softw. Eng.* **2016**, *12*, 109–140. [[CrossRef](#)]
18. Fernandez, E.B.; Yoshioka, N.; Washizaki, H.; Yoder, J. Abstract security patterns and the design of secure systems. *Cybersecurity* **2022**, *5*, 7. [[CrossRef](#)]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.