TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

DIPARTIMENTO DI INGEGNERIA CIVILE E INGEGNERIA INFORMATICA

DOCTORAL DEGREE IN
Computer Science, Control and Geoinformation

Developing Practical Secure Systems: Kernel Solutions for Threat Prevention and Resilience

Tutor:
Prof. **Francesco Quaglia**

Coordinator:
Prof. **Francesco Quaglia**

Candidate:
Pasquale Caporaso

CYCLE XXXVIII
A.Y 2024/2025

Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it.

Donald Knuth

Abstract

In recent years, malware has become easily accessible through the dark web, some of the most dangerous and effective vulnerabilities are sold for millions to unscrupulous organizations, never to be revealed to the public but instead used to disrupt the economic and political ecosystem of entire nations. In the face of such sophisticated attacks, commonly used defense systems struggle, as they cannot easily adapt to an ever-changing landscape of attack strategies. To contrast this rapidly evolving environment, an increasing number of researchers have started to focus their efforts on the development of high-resilience systems which offer additional protection because they are inherently resistant to or capable of rapidly recovering from, cyberattacks.

Sometimes, though, to ensure the absolute security of a system, the designer ignores another important characteristic, its usability. In their book "Security and Usability", Lorrie Cranor and Simon Garfinkel observe, jokingly, that making a computer secure is actually very easy: just turn it off, put it in a safe and throw away the key; this way the machine is definitely secure, just not very usable [69]. This extreme example is used to show that we cannot focus only on security measures, but we also need to consider how those measures impact the user experience.

This thesis focuses on achieving two key objectives: high resilience and usability, by developing solutions that operate at a high privilege level—namely, the kernel level—and can therefore leverage this privilege to modify or extend standard system mechanisms to protect their inner workings and remain transparent to the end user.

The work presented introduces three solutions that operate at distinct

stages of a cyberattack. In the malware analysis stage, we propose *JITScanner*, a tool capable of intercepting and tracing the first execution of each executable page in a monitored program, also considering page modifications and permission changes. *JITScanner* introduces no changes to the monitored program's address space, relying on manipulating the permissions on the page table instead of the usual binary instrumentation, making it suitable for offline malware analysis or dynamic signature generation, as the target application has no easy way to detect the monitoring tool. In the malware detection stage, we introduce *FlowScanner*, an evolution of the concept presented in *JITScanner*, which combines page table manipulation with binary instrumentation to intercept the first access to each basic block of the application, the higher precision obtained by this higher level of granularity makes this tool more suitable for live malware detection where the traced basic block can be compared with a previously obtained signature. Finally, for an already compromised environment, we propose *VaultFS*, a Linux-based file system with associated reference monitor capable of guaranteeing data integrity through the enforcement of a Write-Once Read-Many policy on files, with no specialized hardware required. We will show that this tool remains effective against high-privilege attackers and remains compatible with several real-world use cases.

Evaluation results demonstrate that our solutions achieve strong protection with minimal overhead in both real-world scenarios and standard benchmarks, significantly improving system resilience against both in-the-wild and targeted attacks.

Contents

Abstract **ii**

1 Introduction **2**

1.1 Context 2

1.2 Challenges 4

1.3 Objectives 5

1.4 Contributions 7

1.5 Dissertation outline 8

1.6 Published Works 9

2 Related Work **11**

2.1 General Software Tracing 11

2.2 Malware Analysis and Detection 13

2.3 Data Protection 14

2.4 Summary and Comparison 17

3 Malware Analysis **19**

3.1 JITScanner 22

3.1.1 Baseline Concepts and Methodology 22

3.1.2 Architectural Hints 22

3.1.3 Executable-Page Access Interception 24

3.1.4 Protection Against DoS 31

3.2 Using JITScanner for Detection 33

4 Malware Detection **36**

4.1 FlowScanner 38

4.1.1 Fast Disassembler 39
4.1.2 Execution Manager 41
4.1.3 Management of In-use Page Copies 45
4.1.4 Management of Master Page Copies 47
4.1.5 Dynamic Check Engine 49

5 System Resilience **51**
5.1 VaultFS 52
5.1.1 Threat Model 55
5.1.2 Baseline Concepts and Architecture 57
5.1.3 The Vault File System 60
5.1.4 The Bouncer Subsystem 68
5.1.5 DoS Mitigation 70
5.1.6 Configuration Device 74

6 Experimental Evaluation **77**
6.1 Compatibility 77
6.1.1 VaultFS 78
6.2 Performance 80
6.2.1 VaultFS 80
6.2.2 JITScanner 87
6.2.3 FlowScanner 91
6.3 Effectiveness 94
6.3.1 VaultFS 94
6.3.2 JITScanner and FlowScanner 95

7 Conclusions **100**

8 Acknowledgments **103**

1. Introduction

1.1 Context

The growing sophistication and accessibility of cyberattacks in recent years has drastically expanded the threat landscape faced by modern computing systems. Vulnerabilities—often traded in underground markets for substantial sums—are increasingly leveraged to compromise critical infrastructures, disrupt essential services, and cause wide-scale economic or political damage. Traditional defense mechanisms, while effective against well-known attack patterns, struggle to adapt to techniques that evolve rapidly and often operate at low levels of the system stack. As a result, researchers have shifted considerable attention toward the design of *high-resilience systems*: systems that are not only capable of detecting attacks, but can also inherently withstand and recover from them.

However, security does not exist in isolation. As highlighted by Cranor and Garfinkel [69], the pursuit of perfect protection often comes at the expense of usability—a system that is theoretically secure but practically unusable is ultimately ineffective. Balancing these two dimensions remains a central challenge, particularly when security mechanisms introduce system-wide policies or require specialized hardware support.

There are several approaches one can take to protect a system. In general, we can operate at different stages of the attack timeline by attempting to:

1. Prevent the attacker from gaining unintended privileges on the machine;

2. Block a privileged user from executing malicious programs or commands;

3. Prevent the malicious program from damaging the machine;

Item 1 is the strategy generally used by firewalls and IDS systems [1, 79, 2, 87], which monitor external communication and block messages that violate standard system operations. Alternatively, one could use vulnerability detection techniques such as fuzzing [19, 43, 30, 113] or static code analysis [47, 40, 96] to ensure that, regardless of the attacker's actions, there are no unintended paths to privilege escalation.

Item 2 is the most common approach employed on general-purpose computers. It consists of analyzing and tracing all applications on the machine and identifying patterns of characteristics or behaviors that distinguish benign programs from malicious ones; once this distinction is made, the latter can be terminated, hopefully before any harmful action is performed. This strategy has been extensively studied in the scientific community and involves several components. First, we have works that study how to extract meaningful information from an application—either statically, without running the executable [78, 73, 21, 8] , or dynamically, by collecting information from live applications [97, 57, 3, 99] or by running applications in a controlled environment [16, 112, 50, 51]. Second, we have research on algorithms capable of interpreting the recovered information and classifying applications as either benign or malicious. This can be done using various approaches, ranging from score-based methods [111, 28] to machine learning techniques [68, 13].

Item 3 is a standard approach for industrial systems, where stricter procedures and narrower use cases allow for the limitation of features even for high-privilege users. Examples include reference monitors [91, 39], which can block damaging operations by default at several levels; read-only hardware support, which can protect user data from malicious or accidental deletion [4, 22, 110, 94]; and system-wide debloating solutions, which remove all unnecessary features from live machines [86, 59].

This thesis focuses on items 2 and 3, as these are the areas most under the control of the operating system and where we find that the state of the art can be improved.

1.2 Challenges

The detection of malware and the resilience to their operations represent two complementary strategies for securing modern systems; each encounters substantial challenges that hinder their practical deployment and long-term effectiveness.

Approaches based on application tracing and behavioral analysis must first confront the complexity of modern software itself. Extracting meaningful information from applications is far from straightforward: static analysis is impeded by code obfuscation, self-modifying code, and the use of dynamically loaded components that remain invisible until runtime [11].

The more popular approach is live dynamic analysis which, while valuable for understanding real-time behavior, still suffers from several fundamental limitations that affect both its effectiveness and safety. To begin with, user-level analysis techniques such as API hooking, dynamic binary instrumentation (DBI), and similar mechanisms operate within the same privilege space as the malware. This makes them highly susceptible to evasion: sophisticated samples routinely detect instrumentation frameworks and unhook critical functions to bypass user-space monitoring entirely [11, 5]. When analysis is moved into the kernel for increased robustness, the problem becomes visibility, kernel-level monitoring typically captures only high-level events such as system calls, offering limited insight into the fine-grained, user-space execution logic where many malicious behaviors originate. As a result, analysts see only the endpoints of complex actions rather than the detailed steps leading up to them.

Furthermore, attempts to compensate for these gaps with periodic memory scanning introduce their own challenges. Such scans impose heavy performance overhead, especially when performed frequently enough to increase the likelihood of catching transient malicious states. Even then, the approach provides no guarantee of capturing meaningful snapshots: modern malware often employs just-in-time decryption, multi-stage unpacking, and re-encryption during execution, ensuring that any scan may encounter only obfuscated or

inert data. This combination of high cost and low reliability limits the practical utility of memory-based inspection. More critically, the very nature of live dynamic analysis carries inherent risk because the malware must be executed to be observed, it has an opportunity to perform harmful actions, such as system modification, data exfiltration, or lateral movement, before its behavior is detected or sufficiently contained. In high-stakes environments, this risk can outweigh the benefits, underscoring the need for safer and more controlled analysis methods.

Finally, capability-limiting and damage-prevention mechanisms shift the focus from detecting malicious activity to constraining what actions can meaningfully harm the system. While industrial settings can leverage strict workflows in controlled environments, several challenges arise when trying to apply these techniques more broadly. Reference monitors and fine-grained policy systems require precise, manually defined rules [92] that must remain synchronized with evolving software stacks, creating significant maintenance burdens and increasing the risk of misconfiguration. Read-only hardware support and similar physical protections offer robust safeguards, but are costly to deploy at scale, require specialized hardware not always available in legacy systems, and often restrict legitimate operations such as software updates or system maintenance. Debloating techniques promise to reduce attack surface by removing unnecessary functionality, yet determining which components are truly unnecessary is itself a complex task; overly aggressive debloating risks breaking compatibility or reducing system usability. Additionally, all these mechanisms must balance their protective capabilities with performance considerations: excessive policy enforcement or sandboxing can introduce latency or impede normal workflows, ultimately discouraging adoption.

1.3 Objectives

This thesis addresses these challenges by proposing kernel-level solutions for monitoring, detection, and enforcement that move beyond current high-

level mechanisms and provide fine-grained control over a program's execution. The goal is to design systems that can intervene directly in a binary's behavior with greater precision than traditional system-call level monitoring, enabling analysts and defenders to obtain a higher level of information in real time. To ensure robust protection, the proposed solutions operate in both synchronous and asynchronous modes: synchronous controls provide immediate action when the kernel detects sensitive events, preventing damage, while asynchronous components can perform more complex pattern analysis without stalling execution. Throughout this design, a core objective is to maintain a practical performance baseline, ensuring that these defensive capabilities introduce only minimal overhead and remain feasible for deployment in real-world environments where usability and efficiency are essential.

In detail, we aim to achieve the following objectives:

1. Present application tracing solutions that can operate with high levels of granularity, focusing on the interception of relevant moments in the program lifetime, enabling the recovery of valuable information without relying on periodic or random monitoring. This tracing should be able to operate in two different modes:

    1a. *Analysis mode*: the tool should recover execution information while being mostly undetectable by the monitored program, this is intended for use in automated sandbox environments for the recovery of signatures or the study of behavioral patterns. A stealthy tool here is required as most malware tends to exit prematurely if they discover the presence of an analysis environment;

    1b. *Detection mode*: the tool should recover execution information while being mostly resistant to evasion, this is intended for use in a live environment, here the tool is used for protection and we do not mind if it is detected as long as the malware cannot bypass its monitoring;

2. Present a reference monitor that can guarantee security, specifically data integrity, in case of detection failure, even in the face of a high-privileged

insider, without requiring a static or complex configuration and without relying on hardware support;

3. Ensure that all solutions cannot be tampered with, i.e, ensure that they operate on a higher privilege level compared to the attack which they are trying to prevent;

4. Demonstrate that, even with the overhead and limitations added by these security features, the system remains usable in standard real-world environments and the observed overhead is minimal;

1.4 Contributions

The contributions of this work are organized around three solutions that operate in distinct phases of an attack life-cycle:

Malware analysis phase: for the analysis objective *1a* we implement *JITScanner*, a tool that intercepts the first execution of each executable page of a program, including pages which have changed permissions or have been modified. *JITScanner* is designed to analyze malware that employs self-modifying or self-patching behavior, a common strategy to bypass static detection tools and by operating at page granularity. We achieve this with a strategy we call *Virtualized Page Permissions*, which manipulates the page table of the monitored program to intercept relevant events, namely the first write and execute accesses on a page, which we use to detect the moment an encrypted malware is copied into memory, which we can then extract or analyze. By operating only inside the page table we are able to maintain the address space of the application untouched as there is no need for binary instrumentation, which greatly lowers detectability. By positioning itself in the kernel we achieve *Objective 3* and by injecting our logic in the page fault interrupt, which is already a costly operation, we achieve *Objective 4*.

Malware detection phase: for the detection objective *1b* we implement *FlowScanner*, a tool that relies on the same page interception mechanisms of *JITScanner*, but expands it to operate at the granularity of the basic block.

By raising the granularity we obtain more refined information on the execution flow of the application which improves the detection rate, however, to achieve this improvement, we require the injection in the address space of illegal opcodes which are then intercepted to detect movement inside the page. This technique allows the malware to detect the presence of *FlowScanner* by simply reading the contents of its address space, meaning that this tool is suited for malware detection due to its high precision but not effective for analysis. Again, by positioning itself in the kernel we achieve *Objective 3* and by intercepting *only the first* execution of each basic block, we achieve *Objective 4*.

System Resilience phase: finally, in case of a failure in the previous two phases, we implement a final defense in *VaultFS*, a Linux write-once file system that provides strong integrity guarantees even in the presence of an attacker with root privileges. VaultFS prevents the modification or deletion of previously written data and blocks unauthorized write attempts even at the block-device level, making it suitable for highly critical environments where data immutability is essential. This achieves *Objective 1* while maintaining *Objectives 3 and 4*

Together, these solutions provide a coherent architectural approach to system resilience: they secure both the initial execution of untrusted software and the long-term integrity of critical data, two areas frequently targeted by advanced adversaries. Importantly, all the solutions are implemented entirely at the software level, require no specialized hardware, and impose minimal performance overhead, making them compatible with a wide range of real-world deployments.

1.5 Dissertation outline

The remainder of this thesis is organized as follows. Chapter 2 surveys related works in the attack phases tackled by each tool we propose in this thesis, namely research in software tracing for detection and analysis and data-

integrity protection. Chapter 3 talks about the malware analysis phase, introduces *JITScanner* and its design and implementation. Chapter 4 presents the malware detection phase with *FlowScanner*, detailing its design and practical considerations. Chapter 5 introduces the system resilience phase, with the architecture, threat model of VaultFS. In Chapter 6 we conduct an experimental evaluation of all solutions presented.

1.6 Published Works

During my PhD, I've been the first author and co-author of the following publications:

- Caporaso, P., Bianchi, G., & Quaglia, F. Vaultfs: Data Integrity Via Write-Once Software Support at the File System Level. *Accepted* at ACM Digital Threats: Research and Practice.
- Caporaso, P., Zarrelli, L., Bianchi, G., & Quaglia, F. (2026). FlowScanner: Malware Detection via Kernel-supported Basic-Block Tracing. *Accepted* at Detection of Intrusions and Malware, and Vulnerability Assessment: 23nd International Conference, DIMVA 2026, Chania, Greece, July 1–3, 2026, Proceedings, Part I. To be Presented at the Chania, Greece. Berlin, Heidelberg: Springer-Verlag.
- Caporaso, P., Bianchi, G., & Quaglia, F. (2024). JITScanner: Just-in-Time Executable Page Check in the Linux Operating System. Applied Sciences, 14(5), 1912. https://doi.org/10.3390/app14051912
- Caporaso, P., Bianchi, G., & Quaglia, F. (2023, August). Jitscanner: Just-in-time executable page check in the linux operating system. In Proceedings of the 18th International Conference on Availability, Reliability and Security (pp. 1-8). DOI: 10.1145/3600160.3605035. **Winner of Best Paper Award at the FARES Workshop**.

- Calavaro, M., Caporaso, P., Bianchi, G., & Quaglia, F. (2025, November). Thwarting ROP Attacks and Unveiling User Level Stack Tampering through Kernel Shadow Stack. 2025 23rd International Symposium on Network Computing and Applications (NCA), 190–199. DOI:10.1109/NCA67271.2025.00040

- Caporaso, P., Bianchi, G., & Quaglia, F. (2024). Benchmarking Virtualized Page Permission for Malware Detection: a Web Server Case Study. In Proceedings of the 8th Italian Conference on Cyber Security (ITASEC 2024), Salerno, Italy, April 8-12, 2024. Retrieved from https://ceur-ws.org/Vol-3731/paper10.pdf

- Manenti, E., Caporaso, P., Bianchi, G., & Quaglia, F. (2025). VFSMon: an Innovative Reference Monitor in Linux. Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), Bologna, Italy, February 03-08, 2025. Retrieved from https://ceur-ws.org/Vol-3962/paper46.pdf

- Bernardinetti, G., Caporaso, P., Di Cristofaro, D., Quaglia, F., & Bianchi, G. (2023, June). PHOENIX: A cloud-based framework for ensemble malware detection. In 2023 21st Mediterranean Communication and Computer Networking Conference (MedComNet) (pp. 11-14). IEEE. DOI: 10.1109/MedComNet58619.2023.10168868

- Calavaro, M., Caporaso, P., Capotombolo, L., Bianchi, G., & Quaglia, F. (2025, July). Poster: On the Usage of Kernel Shadow Stacks for User-Level Programs. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 237-242). Cham: Springer Nature Switzerland.

2. Related Work

In this chapter, we discuss existing technologies in the field of operating system security. In Section 2.1, we examine generic techniques for tracing binary applications, that is, strategies that were not originally designed for analysis or detection but can nevertheless be used for these purposes. Section 2.2 addresses malware analysis and detection and in Section 2.3, we discuss data protection solutions. Finally, in Section 2.4 we discuss where the solutions presented in this thesis fit in the state of the art and which problems they address.

2.1 General Software Tracing

One of the most common techniques for execution tracing is static instrumentation or binary rewriting, as used in tools such as [31, 108, 37, 62, 9]. This approach involves modifying a binary by inserting instructions to extract flow or coverage information during execution, and incurs relatively low runtime overhead, as both analysis and instrumentation occur before execution. However, this approach is mainly applied in software testing or fuzzing, where the target is cooperative and its behavior is well understood. For instance, TinyInst [37] explicitly assumes that "the target is well-behaved", while RetroWrite [31] supports only position-independent code and ignores obfuscation, assumptions that fail in malware detection or analysis, where adversarial behavior and obfuscation are common.

Hardware-assisted execution tracing and debugging have recently become available through technologies such as Intel Processor Trace (Intel PT) and

ARM CoreSight [63]. These solutions leverage built-in processor features to capture and collect information about a program's execution. Direct hardware integration yields low runtime overhead but produces a massive, non-configurable volume of trace data, posing challenges for efficient collection and analysis. Tools like perf [34] attempt to reduce this load by limiting trace duration or restricting tracing to selected cases, but such constraints are unsuitable for malware detection, where full and continuous monitoring is often required. The runtime overhead is actually too great even for analysis, meaning that is impossible to trace the complete malware execution which is often required for precise behavioral analysis or signature generation.

Coverage-guided tracing, used in tools such as [36, 77, 76], instruments only the first execution of each basic-block, allowing subsequent executions to run at native speed. Current solutions insert interrupt instructions (e.g., 0xCC) at the entry and exit of each basic-block. These interrupts are intercepted and handled by an external monitor, which then removes them to restore normal execution. Since this placement is performed statically before execution, it inherits the same limitations as static binary rewriters and is ineffective against self-patching or adversarial programs that can overwrite the interrupts at runtime to evade tracing.

The most widely used tools for generic instrumentation are Dynamic Binary Instrumentation (DBI) frameworks such as QEMU, Pin, DynamoRIO, and R-Visor [14, 70, 10, 52]. These tools execute programs within a user-level interpreter that dynamically rewrites code sections at runtime to insert instrumentation, caches the rewritten sections, and reuses them at near-native speed. To preserve correctness, rewritten code is adjusted for non-PIC memory accesses, due to relocation, and for self-modifying instructions. While highly effective, these tools incur substantial performance overhead because block transitions must still pass through the interpreter to ensure the correct control-flow and determine whether new code needs to be instrumented. This introduces significant overhead, even in cases where no additional instrumentation is required. At the same time, the extensive manipulation of the address

space makes them easily detectable by the monitored programs, rendering them unsuitable for offline analysis.

A tool that lies between static and dynamic instrumentation is Dyninst [12], which combines traditional static binary rewriting with a runtime monitor that can insert or remove instrumentation as needed. Like other static tools, Dyninst offers very low overhead but also inherits key limitations, particularly the lack of support for self-modifying code. Although the runtime monitor can identify modified code regions, implementing detection logic is left entirely to the user. Unlike FlowScanner or DBI frameworks, Dyninst lacks built-in mechanisms for automatically detecting such behavior, making it unsuitable for online malware detection, where self-patching is common and unpredictable. Moreover, similarly to DBI frameworks, it can be easily detected due to the evident instrumentation within the user address space.

2.2 Malware Analysis and Detection

A notable subset of DBI frameworks specifically targets malware analysis, such as SPiDER, SPIKE, and Arancino [102, 29, 83]. These systems employ strategies to achieve near or full-transparency, ensuring the monitored malware remains unaware of the instrumentation and executes normally. However, this is traded with substantial performance overhead, acceptable for offline analysis but impractical for live detection.

Other approaches [16, 112, 50, 51] rely on executing applications within an emulation environment, and aim to utilize shadow memory and metadata to identify and analyze potential malicious alterations in the application address space. Nonetheless, emulation-based methods may introduce varying degrees of interference or opacity during application execution within the emulation environment.

Several studies have addressed malware detection by searching for known signatures in a program's address space. Some approaches focus on identifying signatures within data sections [78] or through their printable characters [73],

while others (e.g., [21, 8]) rely on static analysis to detect malware before execution. In contrast, our objective focuses on signatures within the executable parts of a program, specifically targeting the basic blocks that whichever thread of a process really uses at runtime. Moreover, our approach can identify signatures even when the content of executable pages (or pages made executable during runtime) changes at runtime.

Other approaches detect malicious behavior by intercepting API calls, targeting either library functions [97], as in common antivirus packages [57], or system calls, as in kernel-level solutions leveraging eBPF [3, 99]. Our approach is orthogonal to these methods, as it performs entirely different runtime tasks. Specifically, we identify the exact content of each basic block that is really executed by an application—for performing signature detection—also considering the scenario where the basic-blocks are dynamically rewritten in memory, as occurs with encrypted or obfuscated malware that decrypts or unobfuscates itself at runtime.

Some proposals leverage virtualized page permissions for security purposes [72, 109], detecting the first execution on writable–executable (WX) pages, which are then marked and extracted for analysis. However, these approaches face substantial limitations, as they intercept only the initial execution on a page, leaving subsequent modifications unmonitored and thus ineffective against multi-stage unpacking or complex execution strategies.

2.3 Data Protection

Data Protection has been implemented in the literature according to different approaches, also depending on whether protection involves hot (read/write) or cold (read after write) data.

As for hot data, most works focus on defending against crypto ransomware attacks, with methods targeting the identification of the presence of malware, in order to attempt an early stop of its activity [20, 23, 48, 49, 55, 89, 38, 54]. These works are a subset of the works presented in the previous section, as

here the focus is specifically to detect ransomware, which has some prevalent characteristics which allow for different detection strategies.

One of the main limitations of these works is their long response time, specifically the delay between ransomware execution and detection, which can still result in partial encryption of data. Moreover, although most of the cited solutions include experimental evaluations demonstrating their ability to detect existing ransomware samples, the risk of false negatives in the medium to long term—due to the evolution and adaptation of ransomware techniques—remains a significant concern. Additionally, data alteration attacks that are not caused by crypto-ransomware, but instead result from other attacker actions (e.g., insider threats), are not addressed by these solutions.

Some authors have examined the use of machine learning as a defence based on the identification of the presence of attacker software [42, 60, 61, 93, 100, 103]. These works exploit different features in order to identify the attack, like for example API calls or sequences of opcodes in the software structure. In any case, these types of solutions need time from the collection of information in order to detect the attack. This might still lead to partially corrupted (e.g., encrypted) data on the file system.

An orthogonal approach for guaranteeing data integrity is the one of relying on recovery techniques, which should provide the ability to recompute the original file content even after a data corruption by a malware, this same approach has been implemented both at kernel level [23, 45] and at SSD firmware level [71, 44, 107, 74, 81, 7]. However, even looking at the more advanced solutions, this way of proceeding is still linked to the possibility of false negatives—in fact, data for the recovery support are maintained until threads are still under judgment of their activities, and in any case not beyond a time limit.

Still for data recovery, the solution in [56] is an alternative proposal that the literature offers, in particular against crypto ransomware attacks. It is based on the recording of cryptographic keys managed at the level of the libraries hosted by the operating system, and on the exploitation of these keys whenever files need to be recovered after an encryption by ransomware. The main limit

of this approach is that malware can include cryptographic services that allow bypassing operating-system hosted libraries.

The approach in [32] presents a solution where file system updates are left pending into volatile memory, at the level of the page-cache of the operating system. They are finally flushed to the device (making the original file content overwritten) only after a delay used to more effectively assess if the threads that have written the new file content can be considered non-malicious. This is done relying on the acquisition of statistics related to the thread behavior along a time window. However, this solution can be still affected by false negatives.

As for literature solutions specifically oriented to cold data, we find [98]. In this work, the authors exploit a combination of techniques in order to provide a safe backup service. The cold copies of data (the backups) are kept in a separate partition of storage, and are made accessible exclusively through a virtual machine that is configured to host and run a minimal set of applications and services (those needed for the management of the backups). The main limit of this solution is that backups are anyhow maintained into a file system offering conventional data access support. Therefore the solution is bypassable in an attack scenario based on privilege escalation leading to root-level operations on the file system (and on the partition) devoted to the backups.

Many read-only file systems have been proposed [27, 95]. However, all the solutions we have found rely on having their contents decided at the time of formatting, i.e., before the file system is mounted. This means they cannot support a typical WORM workflow, where files are created and written gradually over time while the system is online.

Versioned file systems [75, 25] maintain on disk a history of file modifications, allowing files to be restored after accidental or malicious deletion. However, no existing solution fully addresses the problem of the insider threat, as most versioned file systems still allow a privileged user to delete the stored history.

Finally, various solutions offering the support for preventing deletion or

change of data, and for meeting regulatory compliance requirements, have been based on WORM device technologies [4, 22, 110, 94], which offer hardware support for this specific purpose.

2.4 Summary and Comparison

JITScanner and FlowScanner are substantially different from existing solutions.

The objective of both systems is to strike a balance between performance, granularity, robustness, and intrusiveness, with JITScanner deliberately sacrificing some granularity in favor of a more stealthy approach, and FlowScanner prioritizing granularity at the cost of increased intrusiveness. No existing solution achieves this balance as effectively as our approach. Static instrumentation and coverage-guided techniques assume cooperative, non obfuscated binaries and therefore lack robustness. Hardware-assisted tracing offers low overhead but produces excessive, inflexible trace data, making it impractical for continuous monitoring. Dynamic Binary Instrumentation and emulation-based systems provide greater coverage and transparency but incur prohibitive performance overhead and are often detectable by the monitored program. Detection-focused malware defenses, such as signature scanning and API or system-call monitoring, rely on probabilistic models or observe only limited execution phases, rendering them ineffective against dynamically rewritten or multi-stage malware. Overall, the solutions presented in this thesis address these limitations by enabling precise and continuous tracking of the code actually executed in hostile environments.

VaultFS differs from existing literature on multiple fronts.
First, VaultFS is fully orthogonal to hardware WORM devices since it works with common read/write devices (e.g. hard disks and SSD), thus requiring no investments at all on specific hardware technologies. Also, as a full software solution, it can operate in both bare-metal and virtual environments, still with no need for hardware specific facilities within the underlying platform. Fur-

thermore, as we mentioned, VaultFS can be configured to manage a protection lifetime, after which the storage blocks originally used for a file are reusable, which is instead not directly allowed with common WORM devices. Hence, it provides a higher flexibility also in terms of actual usage of the storage. Additionally, it offers the support for DoS mitigation, in terms of usage of the device storage by non-trusted applications that simply try to saturate the file system with dummy data. This facility is not the target of WORM compliant storage systems.

Second, existing software-level solutions rely primarily on detection-based mechanisms. As a result, they are designed mainly to counter malware attacks and typically provide no protection against insider adversaries. VaultFS is the first software-only proposal to explicitly consider this threat model. Moreover, even when addressing external attacks, current software solutions depend on detection algorithms, such as score-based systems or machine learning techniques, which, although largely effective, remain inherently probabilistic. Consequently, they cannot guarantee the detection of every attack, but only of a large fraction of them. In the context of critical infrastructures, this limitation is unacceptable, as even a single successful attack out of thousands may cause irreparable damage. VaultFS avoids this issue through a secure-by-default design: data integrity is always enforced, since every interaction with the system, whether benign or malicious, must comply with its immutability rules.

3. Malware Analysis

Malware analysis serves as the first step in the line of defense against a cyberattack, it consists of the study of malware in a safe, sandboxed environment where monitoring tools can be used to extract information. It is during this stage that analysts create signatures, behavioral patterns, and other indicators that can later be used by real-time detection and response mechanisms. Yet this process is far from straightforward: offline analysis is a continuous game of cat and mouse with malware authors, who design their creations to evade scrutiny and hide their true intentions when they detect analysis tools or sandbox conditions. As a result, the tools and techniques used in offline analysis must be as stealthy and resilient as the threats they aim to uncover.

This chapter explores this process and presents *JITScanner* (Just-in-Time Scanner), a Linux-oriented solution which is based on the identification—along wall-clock-time—of suitable points where the actual analysis on the content of a specific executable page should be carried out. The objective is the one of avoiding at all pages that, although belonging to some program—or to some external library which the program uses—are not actually materialized in RAM[1] and/or their content incarnation is not actually accessed for instruction fetch. Therefore, we avoid the cost for analyzing any page that the application is not actually using for keeping machine instructions that are really accessed for their execution.

Importantly, by using page fault events as the interception point for analy-

[1]We use the term “materialized" to indicate that a virtual page is currently present in some page frame at the RAM level, which is accessible by the application in its address space. Otherwise the page is intended as not-materialized, and its access will generate a fault to be managed by the operating system. As an example, pages that in a Posix operating system are `mmap`-ed at some point in time, can be materialized later on along the application execution.

sis, JITScanner performs no modifications to the monitored program's address space, ensuring that the program has no means of detecting that it is being traced.

From a technical perspective, JITScanner encounters a significant challenge with pages that are allocated for Write-Execute (WX) usage. Simply collecting these pages at the time their content materializes and upon their initial use for executing machine instructions is inadequate for confirming that they will not harbor exploitable malicious signatures accessible to attackers. At the same time, it is crucial to note that WX pages hold substantial relevance in legitimate scenarios, particularly in supporting Just-in-Time (JIT) compiling within language-specific virtual machines [46].

To address this issue, our solution incorporates support for a security-focused state machine, managed at the kernel level. This "shadow" state machine operates in a way that the actual actions an application can perform on any WX (Write-Execute) page—such as writing or fetching instructions—are entirely logical. In particular, at any given moment, only one of the two possibilities (W or X) is allowed to happen without being traced by the operating system kernel.

This approach enables us to identify instances in which an executable page is revisited by the CPU to fetch machine instructions after its content has been updated. Consequently, it allows us to intercept the precise moment at which new potentially malicious signatures can be identified, following the modification of the page's content, that is, the re-materialization of its content.

Importantly, this solution also addresses scenarios where an attacker exploits encrypted code that is later decrypted over time into an executable page. This particular scenario is critical and cannot be effectively handled by traditional static analysis techniques and tools and it presents a potential loophole for evading antivirus software that conducts dynamic analysis of active programs [11].

In our solution, we also provide the support for analyzing the page content upon an instruction fetch if a previously materialized non-executable page is

requested to become executable via specific system calls. The same applies to executable pages that are also made writable and undergo updates over time.

By incorporating this functionality, we ensure that whenever a non-executable page is dynamically altered to become executable or when executable pages are permitted to be writable and are modified, our system analyzes their content during instruction fetch.

The architecture of JITScanner includes both synchronous analysis, which happens in the kernel, and a subsystem to allow the communication of executable-page snapshots to user level demons, which can perform the page-content check asynchronously, off the critical path of the applications under control, here the page content can be saved for the later generation of signatures.

This design can also enable JITScanner to be used as a detection engine. For example, in a live operating environment where performance overhead must be minimal, JITScanner can be deployed using a two-level detection scheme. First, in the kernel, the synchronous component can search a small database of critical signatures or perform quick statistical analyses. Second, asynchronously in user space, the system can perform a complete signature check and other more complex operations using the page snapshot. Both components can then decide to kill the monitored application, either before the malicious signature is executed or immediately afterward.

Overall, the transparency of JITScanner—namely, the fact that it operates without modifying the monitored program's address space and is therefore not detectable—enables its use as a offline analysis tool, since potential malware has no indication of being monitored and thus exhibits its malicious behavior. At the same time, its low performance overhead makes JITScanner suitable as a live detection mechanism, as it can monitor applications without significantly impacting system performance. Additional considerations regarding the optimal use of JITScanner are discussed in this chapter, particularly in Section 3.2.

3.1 JITScanner

3.1.1 Baseline Concepts and Methodology

The design principle of JITScanner is simple yet effective: we intercept and extract an executable page *only when that page content is actually materialized in memory and is accessed for an instruction fetch.* This approach detects malware that decrypts malicious stages during the execution, and thwarts malicious code which delays execution to evade sandboxing.

The malware analysis conducted by JITScanner is based upon basic principles, specifically the reliance on page-fault handlers for intercepting and managing the CPU fetch of instructions from executable pages. Although JITScanner is the first solution to leverage this mechanism for the purpose of conducting memory analysis on Linux systems—while also including WX pages—the ideas it relies on are essentially reusable in any other operating system family. Overall, the methodology JITScanner exploits lies naturally on contemporary address space management solutions in common software systems. We believe this widens the applicability of our ideas.

In the following sections, we delve into the details of the architecture and technical approaches that underpin the design of JITScanner.

3.1.2 Architectural Hints

The objectives of our system are two-fold: we want to perform a comprehensive analysis of every executable page that can be exploited by an execution flow of a program, while minimizing at the same time the intrusiveness. To achieve these objectives, our system employs a multi-faceted approach, whose scheme is summarized in Figure 3.1.

The architecture of our solution exploits both a kernel-level layer and a user-level layer. The kernel-level layer, which is fully implemented as a Linux Kernel Module (LKM), performs two critical actions:

1) it intercepts the first access to an executable page (or an updated exe-

cutable page—this is the case of WX pages), which is carried out via an instruction fetch by the CPU;

2) while handling the interception of the instruction fetch, and after the kernel materialized the target page in RAM, we can perform any critical operation on the page content, including signature checks in a malware detection scenario, which might lead to the forced termination of the application.

The verification process described in point 2) occurs within the same thread that initiated the access (for instruction fetch) to the executable page. Consequently, it operates synchronously concerning the application's execution.

To ensure swift handling of this scenario, allowing the thread to quickly resume executing user-level code if the application termination is not prompted, our architecture integrates modules responsible for conducting these critical operations directly at the kernel level. However, if used in a live operating environment for detection, it is essential to restrict the number of checks performed synchronously. For instance, a common scenario might involve merely verifying if the page content includes the embedding of shellcode. Limiting the synchronous checks helps maintain efficient and responsive execution of the application by balancing the critical verification tasks without impeding the performance of the user-level code.

To still enable a more ample analysis of the page content, we exploit the user-level part of the architecture.

It involves a program that receives input from the kernel in the form of a tuple: $< page\_content, offset, process\_id, thread\_id >$. The $offset$ denotes the position (linear address) of the first fetched instruction within the page. This program conducts a more comprehensive analysis of the page content asynchronously concerning the thread that initiated the page access. This approach allows for a broader examination to extract potential malicious signatures within the executable page, carried out off the critical path of the original application's execution. When operating as a detection engine, this program can be configured to take action, such as blocking the application

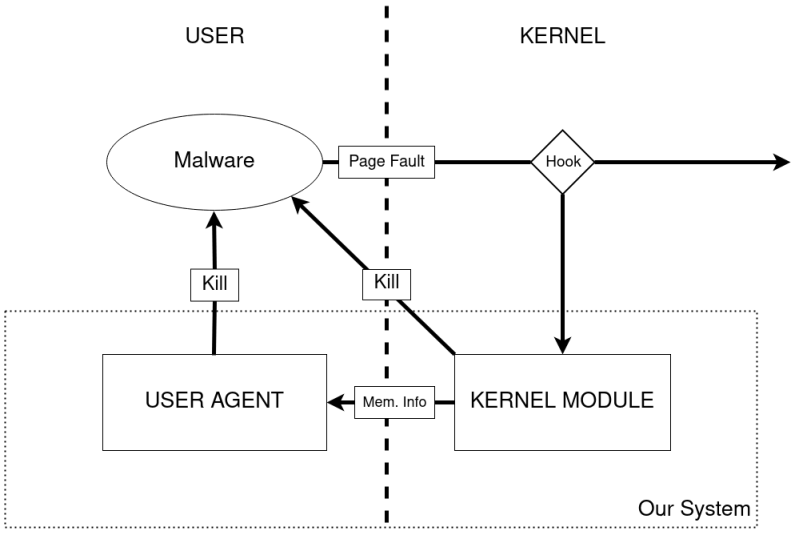

Figure 3.1: System Architecture

(e.g., via the `kill()` system call), if it identifies any security-critical aspect during the asynchronous check.

Moreover, the *page_content* included in the tuple represents a snapshot of the executable page captured at the time of the intercepted instruction fetch. This snapshot capability enables the analysis of the complete history of the content of any WX (Write-Execute) page over time, a feature not facilitated by solutions like [72]. This historical analysis provides a more comprehensive view of changes occurring within WX pages, enhancing the detection and understanding of potential security threats.

Our implementation has been tailored for kernel version 6 of Linux, and for x86 processors with PAE (Physical Address Extension) enabled. However, generalizing our LKM to support other architectures should be straightforward since we have utilized generic kernel functions wherever possible.

3.1.3 Executable-Page Access Interception

Within JITScanner's framework, the primary focus lies in intercepting a genuine access to an executable virtual page, particularly when the CPU actively fetches an instruction from that specific virtual page. Consequently, our solution does not initiate checks on any executable virtual page until the moment when the application actually requires the binary code contained within

it.

In line with the typical on-demand paging process, this need arises after an executable page is allocated by the kernel and placed into a RAM frame. Subsequently, the page-table of the application is updated, granting the execution capability and making the page accessible. It is precisely at this point—when the CPU fetches instructions from the executable page—that JITScanner intervenes to conduct the necessary checks and verifications, ensuring the integrity and security of the fetched binary code.

To precisely identify the moment when a memory page is initially accessed for an instruction fetch, our solution employs a series of strategically placed kernel probes within the page fault handling process. Specifically, we utilize the kprobe subsystem provided by the Linux operating system to set up a hook on the return of a core page-fault handling procedure. In greater detail, we have installed a kretprobe on the `handle_mm_fault()` procedure within the Linux kernel. Upon completion of this procedure's execution, the target page that triggered the fault has already been materialized in RAM. At this stage, we can conduct the synchronous check on its content since it is now available in memory. Additionally, we take a snapshot of the page, which is managed for subsequent examination by the asynchronous check. This approach allows us to precisely time the execution of checks and efficiently manage the content of accessed pages for both synchronous and asynchronous analysis.

An essential consideration in our hook execution is to perform the page check only if the accessed page for instruction fetch is a legitimate page within the address space. Any attempt to access a non-legitimate page doesn't result in the actual materialization of the page in RAM through the operating system's `handle_mm_fault()` function. Instead, it leads to the delivery of the `SIGSEGV` signal to the application, indicating a segmentation fault or an invalid memory access attempt.

This distinction is particularly pertinent in our management of WX (Write-Execute) pages. For these pages, an "original" page fault triggered by the operating system—associated with the first access to the virtual executable page,

which we intercept using the hook—is insufficient for checking the absence of malicious signatures. As a result, careful consideration is necessary to ensure that our page checks are executed only when the accessed pages are legitimate within the application's address space, thereby optimizing the effectiveness of our security measures while avoiding unnecessary checks on non-legitimate pages

For managing WX (Write-Execute) pages, we've developed an innovative kernel-level mechanism that dynamically adjusts the actual page permissions while the program is executing. Despite these dynamic modifications, the page remains "logically" accessible at the application level in both Write (W) and Execute (X) modes.

This kernel-level facility operates entirely transparently to the actual program, ensuring that the application is unaware of these permission modifications. However, it grants us the capability to monitor and track page access—whether for writing data or fetching instructions—throughout the program's execution.

Essentially, this facility establishes a shadow state machine that aligns with the legitimate permissions granted to the application for the page. By dynamically managing and tracking the permissions without the application's knowledge, we can maintain security while ensuring necessary access to the pages, offering a nuanced and transparent approach to managing WX pages.

The shadow state machine within our system implements a time-separated alternating mechanism between write and execute permissions for pages. When a page is in "write mode" in the shadow state machine, it allows page-update operations without triggering user execution-mode interruptions such as page faults or interception via our kernel-level support. However, any attempt to execute instructions from that page will result in a fault, redirecting control to our kernel-level hook for interception and handling. Conversely, when a page is in "exec mode" within the shadow state machine, it permits instruction-fetch operations. In this mode, attempts to update the page will trigger a fault, intercepted by our kernel-level support for handling. This alternating

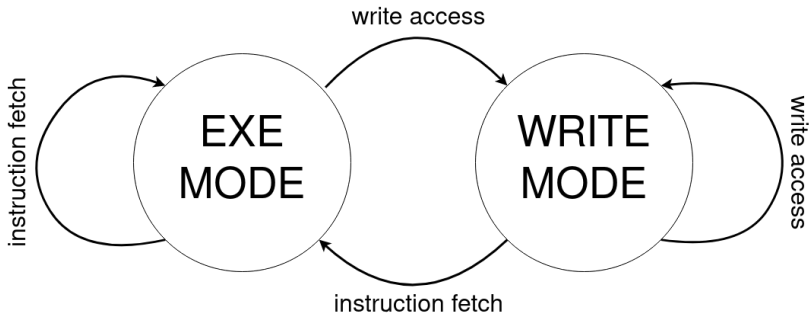

Figure 3.2: Shadow state machine for WX permissions

logic between write and execute permissions ensures a controlled and secure mechanism for managing page access and operations within the application's address space. A representation of this logic is shown in Figure 3.2.

With our facility that manages page permissions dynamically, we have the capability to trigger the page analysis process whenever the access mode transitions between write and execute permissions. However, based on the goals of our solution, we believe that conducting the analysis on the executable page is essential only during instruction fetches. This is the critical moment where potential exploitation of malicious code could occur, making it the prime instance for initiating the signature generation or detection.

By specifically targeting the instruction-fetch operations for page analysis, we can efficiently focus our security measures on the most vulnerable points in time, ensuring that checks for malicious signatures are conducted precisely when the executable page is being accessed for instruction execution. This targeted approach enhances the effectiveness of our security measures while optimizing performance by avoiding unnecessary checks during other access modes.

To support shadowing, we set the XD bit on the lowest level of the page table (the PTE), disabling instruction fetches from the page as soon as the WX page is materialized. By doing this, when the user program tries to execute the instructions on the page, a page fault occurs due to the lack of required permissions. We intercept this page fault and do the following:

- clear the XD bit, to allow the user to execute the page from now on;

- clear the W bit, to disable writes on the page;
- suppress the `SIGSEGV` signal generated by the invalid access to prevent the kernel from terminating the user program or making it run a `SIGSEGV` handler.

To handle the interception of kernel-level functions responsible for delivering the `SIGSEGV` signal during CPU fetches, our approach involves suppressing this signal since the page fault is generated by our shadowing mechanism. In Linux, faults occurring during instruction fetches are managed within the architecture-specific code of the kernel before calling the `handle_mm_fault()` function. However, these functions cannot be directly hooked using the kprobe mechanism. As an alternative, we have chosen to place a kernel probe on the `force_sig_fault()` function, which is triggered whenever an invalid memory access occurs. By hooking into this function, we can effectively suppress the delivery of the signal. Moreover, the synchronous checking of page content and the passage of page snapshots to the asynchronous checker are handled through the hook installed on the return of the `handle_mm_fault()` function. This comprehensive approach enables us to intercept and manage the necessary signals and functions, ensuring the seamless operation of our shadowing mechanism and the subsequent verification processes.

To effectively handle the hook for the `force_sig_fault()` kernel function, it is crucial to differentiate between standard invalid-access faults and those triggered by our shadow state machine. For this purpose, we utilize two unused bits within the lowest level page table entry (PTE) of the `x86` processor. These bits store the original write and execute permissions for a specific page before any modifications by our module.

During the interception of a fault, we check these bits within the PTE. If the fault originates from a genuine invalid access generated by the program and not by our shadow state machine, the bits help us identify this distinction. In such cases, we allow the kernel to handle the fault without any intervention from our module. This mechanism ensures that our system effectively identifies and manages faults triggered by the shadow state machine while allowing standard

invalid-access faults to be handled conventionally by the kernel.

We present in Listings 3.1 and 3.2 the main parts related to the hook-based page management process. In Listing 3.1, we illustrate the management of our state machine in the `handle_mm_fault()` hook, which is invoked on page materialization and a write access violation. The steps are as follows: first, we do some general checks to ensure that the fault occurred in a valid `vm_area` with the appropriate flags. We then examine our `ORIG_WRITE_BIT` bit in the page table, to verify if the page has been modified by our state machine and if writes were initially permitted, if so, we disable execution and allow the write. Otherwise, we let the kernel handle the fault conventionally. In Listings 3.2, we report the hook for `force_sig_fault()`, which is called on an execute-address violation. Here we check if our `ORIG_EXE_BIT` bit is set in the page table. If it is not set, we let the kernel handle the fault. If it is set, this fault is generated under our state machine and needs to be resolved. To this end, we first run a synchronous analysis of the page content at kernel level. When performing detection, if a threat is detected, we allow the kernel to complete the fault, resulting in the delivery of the `SIGKILL` signal to the running process. If no threat is found during the kernel-level check, we transfer the page snapshot to the user agent for asynchronous checking and then fix the page table to allow the execution to continue.

The necessity of a TLB (Translation Lookaside Buffer) flush within the code listings is crucial due to the way TLBs maintain information based on the previous access permissions of a page. When the permissions of a specific page are modified, it becomes imperative to eliminate this page's address from the TLB across all CPUs involved in executing threads operating within the same address space as the intercepted thread accessing memory.

In our implementation, we've leveraged the `__flush_tlb_one_user` API provided by the Linux kernel for this purpose. This API allows us to perform a TLB flush for a specific page address. Additionally, we combine this API with the Inter-Processor-Interrupt (IPI) approach to execute the function across all CPUs involved. This approach ensures that the TLBs across all relevant CPUs

are updated and purged of the outdated page information, aligning them with the modified access permissions for the page.

```
if (general_checks() == OK &&
    pte[fault_page] & ORIG_WRITE_BIT) {

    set_bit(fault_page, XD_BIT);
    set_bit(fault_page, W_BIT);
    set_bit(fault_page, ORIG_EXE_BIT);
    flush_tlb(fault_page);
    goto allow_write;
} else {
    goto standard_fault_handler;
}
```

Listing 3.1: The hook on `handle_mm_fault()`

```
if (pte[rip_page] & ORIG_EXE_BIT) {
    if (sync_check(rip_page) != OK) goto send_sig_kill;

    transfer_page_to_user(rip_page);

    clear_bit(rip_page, XD_BIT);
    clear_bit(rip_page, W_BIT);
    set_bit(fault_page, ORIG_WRITE_BIT);
    flush_tlb(fault_page);
    goto allow_normal_execution;
} else {
    goto standard_fault_handler;
}
```

Listing 3.2: The hook on `force_sig_fault()`

Finally, installing hooks on system calls like `mprotect()` is a significant step within our system. This particular system call is capable of altering the access permissions to a page, such as adding executable (X) or writable (W) permissions to a page that previously had different permissions. By placing hooks on system calls like `mprotect()`, we ensure the correlation between any change in access permissions and the need to recheck the page content upon

the next instruction fetch from that page. This approach guarantees that whenever there is a modification in permissions using `mprotect()` or similar calls, our system is aware of it. Consequently, it can plan to perform necessary content rechecks when the page is accessed for instruction fetching, aligning with the modified permissions and ensuring security integrity.

3.1.4 Protection Against DoS

It is integral for our system that the page fault hook and the driver, responsible for transferring page content and metadata to the user-space agent, communicate efficiently. In our architecture, we employ a hash table for this purpose. This hash table organizes intercepted pages, creating copies (page snapshots) and categorizing them based on their associated process ID. These pages are stored within different buckets in the hash table, allowing parallel access for both insertion (storing a copy of an executable page to be checked) and deletion (transferring the page copy to a user-space buffer).

However, a critical aspect of this solution is the potential for a flow of page-fault interceptions, potentially caused by a malicious program intending to disrupt our page-management system. This influx of page faults can result in uncontrolled effects due to the intensive memory usage required to store copies of executable pages in the kernel-level hash table before transferring them to the user-space agent.

This scenario of numerous intercepted page faults can lead to an excessive volume of memory usage within the kernel, potentially impacting system stability and performance. It is crucial to develop mechanisms or safeguards within the system to handle such scenarios efficiently, ensuring that the system remains resilient against potential attacks while maintaining its stability and operational integrity.

To fortify the system against Denial of Service (DoS) attacks stemming from memory depletion due to excessive kernel-level allocations, we have integrated an additional hash table. This secondary hash table maintains support for parallel access across different buckets. Each entry within this hash table is

responsible for tracking the user ID associated with pending executable-page copies awaiting delivery to the user-level agent, along with a counter representing these pages.

As a preventive measure, we have implemented a threshold mechanism within the system. When the counter surpasses a specified threshold value for a particular user's pending page copies, it signifies an abnormal influx of page faults. In response, actions are taken to prevent potential DoS scenarios. The system can be configured to either terminate or block the process generating additional page faults on behalf of the same user.

This configuration flexibility is achieved by exposing a parameter within our software architecture, accessible through the /sys file system. This level of configurability enables dynamic adjustments based on the system's operational needs and the criticality of the protected environment. By providing this flexibility, administrators can fine-tune the system's response, choosing between process termination or blocking based on the current circumstances and security requirements.

Our system implements a Time-to-Live (TTL) mechanism as a form of rate-limiting control to prevent excessive page copies from being generated for the same user ID. Once an event (kill/block) occurs due to surpassing the threshold, the corresponding entry in the hash table associated with that user enters a TTL period.

During this TTL period, the counter within the hash table entry is decremented each time a copy of an executable page related to the same user ID is successfully delivered to the user-space agent. However, if processes from the same user generate additional page faults requesting page copies during this TTL period, they are still subjected to termination or blocking. Essentially, no new page copies are allowed for processes of that same user throughout the entire TTL duration.

Also, to contrast DoS caused by excessive memory usage of this second hash table, we added a second TTL. If the entry of the hash table associated with a given user ID keeps a counter that remains set to zero for the whole

TTL, then this entry is removed from the hash table—it will be reinserted in the future upon a page fault interception for a program run from the same user ID. This avoids keeping useless entries in this hash table thus contrasting attacks that can be based on the exploitation of programs with the setuid flag active. In fact, these would allow an attacker to operate via multiple user ID values, enlarging at some point in time the number of entries of the hash table.

3.2 Using JITScanner for Detection

The lower overhead achieved by coupling most operations with the page fault handling mechanism, together with the separation of responsibilities between the synchronous kernel module and the asynchronous user-level component, makes JITScanner not only highly performant in analysis scenarios but also usable as a detection engine.

An immediate strategy that can be adopted is to leverage an existing database of previously identified signatures and use JITScanner to search for them in new, previously unseen samples. If a signature is found, the program can be terminated either directly in the kernel or by invoking the `kill` system call from user space.

We also note that performing the kill of the malicious application in JITScanner is a baseline choice, but alternative solutions for the handling of the malware program can be simply adopted. As an example, there are works that have proposed the dynamic setup of facilities at the operating system level in order to manage malware suspicions in specific programs (see, e.g., [18]). The combination of JITScanner with these solutions can be immediate since, rather than killing the identified malware, JITScanner could simply alert an external system (passing to it baseline information, like for example identifiers of processes and users) in order to let that system start the management of the malware according to its own rules.

In the JITScanner architecture, the user-level component can effectively utilize existing code analysis methods accessible in the literature. In our cur-

rent implementation, we have developed a user-level layer centered around Yara rules, a well-established system renowned for malware classification [106].

In any case, the check facilities within our solution can be perceived as plug-ins that are highly extensible and can be effortlessly expanded at any given moment. Fundamentally, our solution comprises an engine dedicated to the management of executable pages within an address space. This engine serves as a stable core foundation upon which various checking mechanisms can be built, refined, and enhanced over time.

This modular architecture aligns with our primary research objective, which revolves around providing an effective kernel-level system to combat dynamic loading and obfuscation techniques. These techniques often pose significant challenges to traditional static analysis methodologies. By focusing on the core engine for managing executable pages and offering a flexible framework for diverse checking mechanisms, our solution aims to address these challenges and ensure an evolving and robust defense against dynamic loading and obfuscation techniques commonly utilized by malicious actors.

We explored the potential usage for detection during the experiments described in Chapter 6, where we configured JITScanner to search for signatures in real malware samples. Although the approach proved effective and achieved a high true positive rate, the tool suffers from low granularity. Specifically, because JITScanner must check the entirety of a memory page for the presence of a signature, it requires an extensive knowledge base, that is, a comprehensive signature database, to be effective, particularly in order to avoid false positives.

This limitation stems from the fact that most applications execute only a small fraction of the code allocated in memory. As an example, Figure 3.3 reports the average and maximum number of bytes actually fetched per page across approximately 600 applications from the "/usr/bin" directory. As shown, the median application executes about 500 bytes per page on average, implying that JITScanner examines, on average, an area roughly eight times larger than what is actually executed. This significantly reduces its precision.

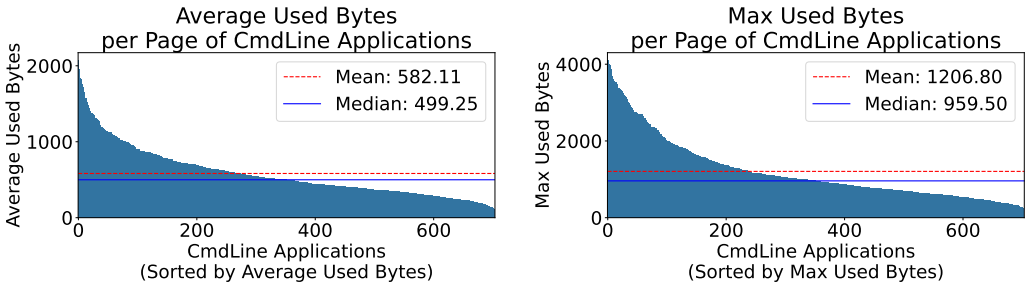

Figure 3.3: Per-page byte usage by CPU-fetch across ~600 `/usr/bin` applications

This observation motivated the development of *FlowScanner*, described in the next chapter, which builds upon the concepts introduced by JITScanner to provide a more detailed view of the portions of a program's address space that are actually executed. As shown in Chapter 6, FlowScanner achieves improved results, particularly in reducing false positives, thanks to its higher precision; however, this improvement comes at the cost of increased intrusiveness, making the tool unsuitable for analysis scenarios.

4. Malware Detection

The most common and widely adopted approach to combating malware is detection, i.e. the development of solutions capable of identifying and stopping malicious programs executing on a system. Dynamic analysis, a prevalent approach, offers advantages over static analysis by enabling observation of runtime behavior and detecting obfuscated or encrypted code used to evade detection. However, executing programs within a controlled environment can be resource-intensive, often necessitating compromises, such as limiting sandboxing to an initial period.

In this chapter, we present *FlowScanner*, an alternative method for dynamic executable detection. *FlowScanner* employs a tracing method that focuses on intercepting the first executable access to a code section at the level of the basic block, when this interception occurs the block can be compared synchronously against a database of signature or saved for later for a more complex analysis. A core characteristic of this approach is that the access is intercepted again if the underlying code changes at runtime. This feature is crucial, as malware often employs self-patching or self-modifying behavior to evade detection mechanisms.

FlowScannr is designed to *simultaneously* achieve three critical objectives: *(O1) low runtime overhead; (O2) fine-grained tracing at the basic-block level; (O3) robust detection of dynamically modified code.*

Achieving all three goals in a single system is fundamentally challenging. Traditional DBI engines support fine granularity (O2) and code tracking (O3), but their tight coupling between application and interpreter logic incurs high runtime costs (violating O1). Conversely, native-execution solutions offer bet-

ter performance (O1) but operate at coarse granularity (violating O2). FlowScanner resolves these trade-offs through a novel in-kernel tracing mechanism based on three key ideas:

1. **Execution-triggered detection** - FlowScanner avoids pre-instrumentation by replacing untouched basic blocks with illegal opcodes. When a thread first fetches such a block, the resulting fault is trapped by the kernel, allowing precise block-level detection *without any interpreter in the execution path*. This enables native-speed reuse of previously seen code (fulfilling O1).

2. **On-demand, block-level scanning** - Signature checks are performed only on the specific portion of the block being fetched, rather than on entire executable pages. This improves detection precision by reducing the false-positive rate typical of page-level systems (fulfilling O2).

3. **Virtualized page permissions** - FlowScanner uses a shadow state machine to track write/execute status of pages, thus detecting/re-validating blocks modified at runtime by self-modifying/unpacking malware (fulfilling O3).

FlowScanner operates by intercepting page faults triggered when a thread attempts to fetch instructions from a page not yet materialized. Upon such a fault, the system uses a lightweight disassembler to determine the block boundary and rewrites the rest of the page with illegal opcodes. This ensures that subsequent transfers to unseen blocks will generate new faults. The accessed block is then transparently restored, scanned for signatures, and optionally forwarded to a user-space agent for asynchronous analysis, all without disrupting the thread's execution semantics.

A further challenge arises from handling pages with Write-Execute (WX) permissions or pages that undergo changes in their permissions at runtime, such as in self-patching or unpacking scenarios, or when shared writable code is used across processes. FlowScanner addresses this by virtualizing page permissions at the kernel level, enabling fine-grained tracking of write-before-execute

violations and permission changes. When a page is altered, it is treated as a new region and instrumented afresh. As we designed it, this mechanism also works on shared memory pages where one process writes code and another executes it, as commonly exploited in multi-process attack vectors.

In the following sections, we delve into the details of the architecture and technical approaches that underpin the design of FlowScanner.

4.1 FlowScanner

The base concept FlowScanner relies on is straightforward: *when a basic-block is newly executed by the CPU along a thread, the fetch of the first instruction in the block is intercepted at the kernel level and the block is logged/examined*. In this paper, we use the standard definition of basic-block: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [41]. We choose the basic-block as the smallest traceable unit as it offers a good compromise: a) it maintains a relatively small number of machine instructions, likely not impacting too much for their check and b) it still retains sufficient information inside it to construct a valid signature. To correctly identify a basic-block in a page at runtime, we have developed a lightweight kernel-level Fast Disassembler capable of identifying flow-altering instructions. It is described in detail in Section 4.1.1.

To transfer control to the kernel at the right moment when a thread starts the real execution of a basic-block that was not traversed (and checked) before, we have implemented a system, called Execution Manager, which is also responsible for restoring the actual content of pages that are accessible in executable mode, where parts of the binary are substituted with illegal-opcode (ill-op) instructions. An in-depth description of the Execution Manager, which also clarifies how it handles WX pages, and code rewrites on these pages by the application, is presented in Section 4.1.2.

To ensure maximum flexibility, in particular for the checks that the Execution Manager can perform synchronously on the content of the basic-block to

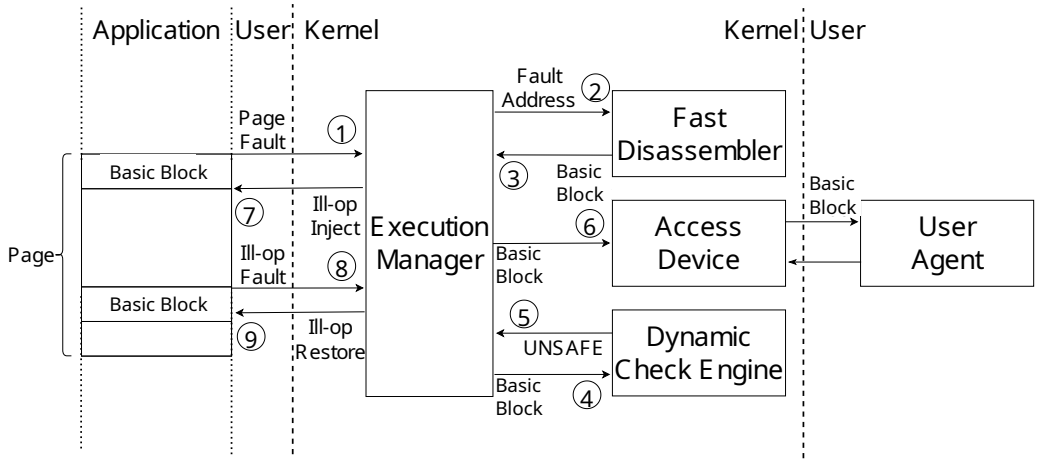

Figure 4.1: FlowScanner System Architecture

be restored upon its access, we have implemented a Dynamic Check Engine, wherein these kernel-level checks can be modified and updated dynamically at runtime, this is described in Section 4.1.5. Also, beyond offering synchronous checking capabilities, FlowScanner supports asynchronous checking, offering an Access Device for user space acquisition of logged basic-blocks.

A complete representation of the entire architecture of FlowScanner, which is made of the ensemble of the aforementioned systems, is depicted in Figure 4.1. Details on how FlowScanner manages pages—like, for example, how it handles shared pages—are provided in Section 4.1.3 and Section 4.1.4. As an additional preliminary note, FlowScanner can work with white/black lists of programs (executable files). Hence its binary analysis can apply to subsets of active processes.

4.1.1 Fast Disassembler

The Fast Disassembler (FD) determines the actual content and length of a basic-block that a thread is entering in, which is located at some point of the address space. FD does not work on the actual page made reachable to the application via the page table. In fact this page may not contain the actual binary code to be inspected for determining the block length—since the page could have been filled with ill-op instructions. FD works on a master copy of the page, not accessible to the application, which keeps the actual page

content. The setup of the kernel-level master copy repository is in charge of the Execution Manager (details are in Section 4.1.4). Hence, when FD is activated, it will receive as input the parameters for reaching the master-copy content to be analyzed for disassembly.

Actually, the task to be carried out by FD poses a unique challenge: given that it is executed synchronously along the same thread that has generated the page-fault or the illegal-instruction fault, the identification of the memory location of the flow-altering instruction that ends the basic-block needs to be carried out at high speed.

To cope with the fastness of the disassembly operation, we adopted a methodology similar to the ones used by length disassemblers [82, 80]. Length disassemblers only need to determine the length of an instruction, they achieve this by dividing the instruction set into classes based on their length. The disassembly result is not a full interpretation of the instruction but rather its correct assignment to one of these classes. In our FD component, we use a similar concept: we maintain length-divided classes of instructions, adding another class composed only of flow-altering instructions. These are the ones we need to fully identify to determine the end of basic-blocks in the address space.

To efficiently put instructions in their associated classes of length, we use a series of 256-bit long bitmaps as look-up tables. During disassembly, each byte of an instruction is used as an index into one or more bitmaps. Depending on the value of the bit found in the bitmap(s) at that index, the instruction is either classified—in terms of its length—or the iteration continues to the next byte of the instruction. Using this approach, we can classify each instruction—in terms of its length—in constant time, a process that is further optimized by the use of bitmaps, as all memory accesses are very compact (the bit-maps have also been aligned to cache-lines) and the classification requires only simple logic operations on the bit-values kept by the bitmaps.

For example, let's examine how the SSE instruction "VADDPS xmm1, xmm2, xmm3" encoded as "C5E858CB" is disassembled:

1. The look-up table for prefixes returns a hit on the byte "C5" as this is

an SSE instruction. By checking the value of the prefix, we identify a VEX prefix which is 2 bytes long, so we skip the first 2 bytes and move directly to the opcode, since these byte values are not needed to identify the length.

2. The look-up table for the opcodes of VEX operations checks if this opcode "58" has no MODRM byte or if it includes an extra IMM8 value. This is not the case here, so we continue to the MODRM byte.

3. For the MODRM byte, we have a series of if-statements to determine the addressing mode. There are look-up tables for more complex modes, but in this case for the byte "CB", we only have registers, so no look-up is performed. We know that there are no more bytes, so the instruction is finished and we assign it to the class of length 4.

The whole process related to this example instruction requires only two look-up operations and a few if-statements. Even more complex instructions do not require significantly more checks, as many bytes are skipped—like in step 1. Moreover, the majority of instructions are simpler than the above example, so the disassembly process remains highly efficient.

4.1.2 Execution Manager

The Execution Manager (EM) is the core component of our infrastructure. It represents the kernel-entry point for the interception of 1) the page-fault—for the case of on demand access to an executable page not yet reachable in RAM—or 2) the illegal-instruction fault—for the case of an access to a zone of the page that has been filled with ill-op instructions by the same EM. Our interceptors of the two different faults, which pass control to FD when the fault occurs (see arrow 2 and arrow 8 in Figure 4.1) are installed by our Linux Kernel Module (LKM) via the kprobe support offered by Linux.

As shown in Figure 4.1, EM is responsible of invoking the execution of tasks in other kernel-level components of the infrastructure, and of providing data to be acquired by the User Agent. Its actions are executed along the critical

path of the running thread—the one hit by the fault—hence we must ensure minimal impact on performance. At the same time, EM is structured in order to carry out its work in a fully transparent manner to the user-level code. It utilizes a two-stage solution—intrinsically linked to the two kinds of faults it intercepts—tailored for working with two distinct memory granularity values. With this solution, it allows discovering *inter-page movement* of threads on different pages of code and also *intra-page movement*, i.e. when a thread starts to execute a new block inside a page that was already accessed for CPU-fetch at a different page-offset.

Such two-step management is not only carried out in relation to executable pages that have been not yet used for instruction fetch by any thread—hence they are on-demand accessed for the first time in the application address space. Rather, it is also carried out, according to a rejuvenation process, for any page that is re-accessed for instruction fetch after it has been updated—for example a page with WX permissions. This is the scenario relevant for FlowScanner, where a (potentially) new code image installed on a page already residing somewhere in RAM is on-demand accessed for the execution of some not yet analyzed basic-block structure. We enter now the details related to both intra-page and inter-page movement tracking that is supported by EM.

Inter-page Movement. To detect the execution flow entrance within an executable page, we exploited a feature present in all modern operating systems known as on-demand paging. According to this concept, when a new memory area is mapped for a process, no new page is immediately transferred to physical memory and/or made accessible through the page table. Instead, this action occurs only upon the first actual access to the page. In the context of executable areas, such access is triggered when the execution flow transits into a new page, resulting in the generation of a page-fault and subsequent materialization of the page in memory. By intercepting these page faults from the kernel—see arrow 1 in Figure 4.1—we effectively capture every initial execution of whatever basic-block in a page. When this page-fault occurs, EM creates a snapshot of the actual page content and saves it into a kernel-level

data structure. We refer to this snapshot as *master copy* of the page (see Section 4.1.4). At the same time, the page that is made reachable via the page table is not the master copy. Rather, it is the one where the only usable basic-block is the one starting at the address that generated the page-fault upon the CPU-fetch operation—all the other parts of this page are set to keep ill-op instructions.

Particular attention has to be paid to processes with areas with WX permissions. On these pages, materialization can occur not during the initial execution, but rather upon the first write access to the page. Consequently, the actual content intended for execution is not yet present on the page when the fault is intercepted and the page is made accessible via the page table. At the same time, WX pages must be monitored due to their widespread use by obfuscation tools such as packers. To address this challenge, FlowScanner employs a shadow state machine, shown in Figure 4.2, permitting only either W or X permissions to be active on the page at any time—even though the application can still use the page according to combined WX permissions. In the shadow state machine, each transition labeled as "Protection Change" is linked to (possible) invocations of the `mprotect(...)` system call, or `mmap(...)` with `MAP_FIXED` flag on an already mapped region, which in our architecture are intercepted via the `kprobe` facility. With this shadow state machine, once the application starts writing to a WX page, its permissions are set to Write-only. Also, the page-table is redirected to the (possibly existing) master copy of the page, which becomes the one actually used. Hence, when this page is accessed by an instruction fetch, a new illegal-access fault is generated and captured by EM via the page-permission shadow state machine—see again arrow 1 in Figure 4.1—and the page is treated as if it was just materialized in RAM. In fact, its actual content, generated by a previous page-write phase, was not present before. In this scenario, EM falls back to managing the page via a newly generated master copy, while again closing (via ill-op) any basic-block different from the one currently accessed. At the same time, when such instruction fetch is intercepted, the EM virtualization of page permissions brings

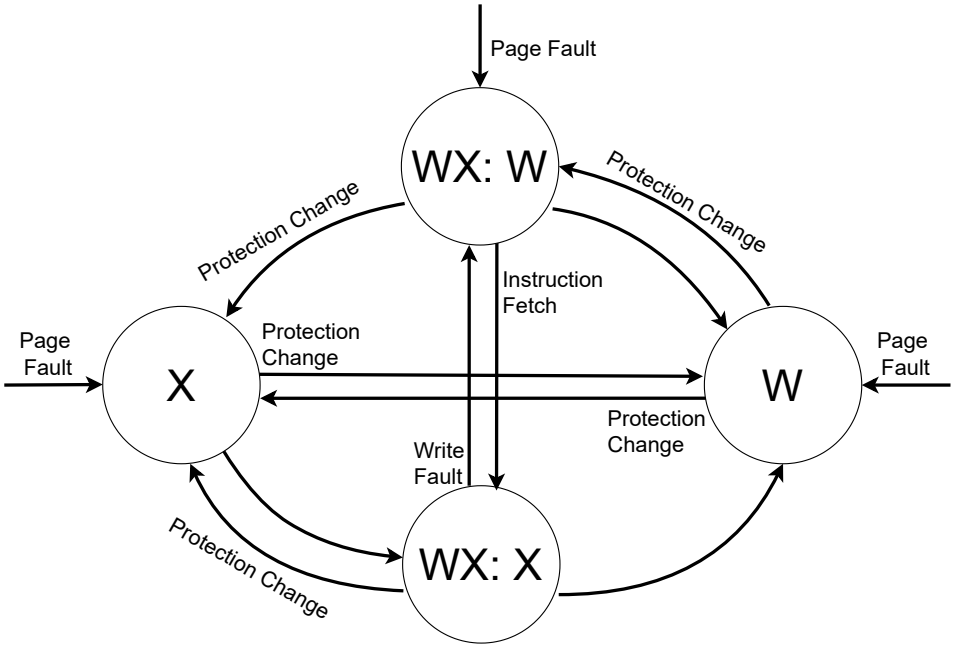

Figure 4.2: Shadow State Machine for Page Permissions

the actual permission to be updated to Execute-only. This ensures that any potential updates to the page are still monitored, as no write operations can occur without generating a fault and passing again through the EM component of our system.

In our implementation, we keep the virtualized permissions for accessing the page into unused bits of the x86 page tables, while using the truly accessed control bits in the page table to grant W or X actual permissions according to the aforementioned alternation, depending on the accesses, compliant to the virtualized permissions, performed on the page.

Intra-page Movement. When handling intra-page execution, we no longer depend on page materialization events and related page table updates, as this is exploitable only upon the initial access to the page. Additionally, we cannot rely on permission changes within the page table, as they cannot separate basic-blocks within the same page. As hinted, at this level, we have devised a dynamic system which relies on the injection of illegal op-codes within the page. Upon accessing a new page for instruction fetch—or a page that was previously updated, like a WX one—we determine the current basic-block contents and

size via our FD component—see arrow 2 in Figure 4.1—which returns the endpoint of the block. Subsequently, we create a copy of the corresponding master page and substitute all the instructions in the original page copy outside the basic-block with ill-op instructions—we refer to this process as *"closing an area"*. The thread that was accessing memory for the instruction fetch will be then allowed to continue, in particular by resuming the execution of the instructions in the basic-block that is outside the closed area in the page.

Any CPU-fetch from another block residing in the same page—which stands in a closed area—will lead to the execution of our injected ill-op instructions, triggering an ill-op fault—see arrow 8 in Figure 4.1. Once again, this is intercepted by the EM component, which passes control to FD, which now parses the master copy of the page and returns once again the end address of the block. Utilizing this information, EM restores the original bytes of the block copying them from the master copy of the page to the actual copy kept in the address space—an operation we call *"opening an area"*—before resuming the execution. This iterative process continues over the entire program lifetime.

It is important to note that signature analysis occurs each time a new block is opened—see arrow 4 in Figure 4.1. Anyhow, once an area has been opened, it remains open indefinitely, as there is no necessity to re-analyze already analyzed parts of the program code for searching signatures.

Finally, we note that the shadow state machine also influences this monitoring level. Upon a page being written to, all areas are considered closed to CPU-fetch (thanks to the shadow state machine managing page permissions) and, as a result, upon execution, they are again subject to analysis.

4.1.3 Management of In-use Page Copies

An initial challenge we encountered while managing the actual copy of an in-use page—the one reachable via the page table—vs the master copy kept at kernel level, revolves around how to close an area, specifically how to write ill-op instructions to an executable user space region lacking write permissions—like a simple X-accessible page. This is a fairly common case as

many executable pages lack write permission and the ones that may have the permission are (virtually) stripped by our shadow state machine. Temporarily altering the permissions of the page to enable writing would introduce a vulnerable window during which an attacker could modify the page code leading to the bypass of the code-fetch interception by FlowScanner—in particular via the elimination of ill-op instructions that FlowScanner places on the page. To address this issue, we incorporated into our Linux module a solution that maps user pages to a new address using the function `kmap_local_page(...)`. With this, the kernel can manipulate the memory freely using the newly mapped address, treating it like any other kernel page, without necessitating changes to the permissions on the user side.

Another special scenario is related to the case where a single basic-block or even a single instruction is divided across two different pages, which we refer to as a "cross-page code". Let's designate page A as the page containing the start of the basic-block (or instruction) and page B as the page containing its end. The state of page A does not impact our handling of this situation, so we assume that we have intercepted the start of CPU-fetch through a newly accessed basic-block at the end of page A, and this block spans the page boundary with page B, a situation easily recognized through our FD component. Now, we need to consider two different states for page B: (1) Page B is already materialized in RAM, but its start is a closed area. In this case, we simply add the actual beginning of page B—retrieved from our saved master copy—to the FD buffer, and restore it up to the next flow-altering instruction identified by FD. (2) Page B is not yet materialized in RAM. In this case, we can initiate the materialization process ourselves by reading the next page and putting it into the table of master-copy pages, also passing it to our disassembler. Then, after the structure of the page-crossing basic-block is determined, we generate in memory the actual copy of page B (accessible via the page table) which is a fully closed version of the master copy, except for this initial basic-block. In both the above cases, any other attempt to move to some closed block inside the page will be managed via the aforementioned inter-page movement

tracking mechanism.

We note that this special treatment of cross-page code is needed to make our FD component always work on instruction-aligned addresses (rather than the start address of a page) when faults caused by FlowScanner—because of the presence of ill-op instructions on pages—take place. Also, this solution enables FlowScanner to take into account basic-blocks that cross pages—in particular for performing signature checks. These types of basic-blocks are not considered by pure page-based scanners in the literature (e.g. [17, 72])—since their signature check operations are limited to the content of a single page. Hence, FlowScanner offers a more precise possibility to check the presence of signatures in the address space of the application, in particular considering really used parts of code.

4.1.4 Management of Master Page Copies

To manage master copies of the pages, we exploited a two-level hash table, initially indexed by PID and then by the address of the page. Each node of the table contains a pointer to a "page_info" structure, which stores the reference to the master copy of the page and metadata on the original page.

The "page_info" structure, and consequently the master copy of the page, can be associated with multiple entries in the hash table, depending on how the original page is shared among the processes in the system. We maintain a reference counter in the metadata, enabling us to track the number of processes utilizing the structure and ensuring a safe deallocation of the information when it is no longer in use. Such management is based on tasks executed by probes we installed on kernel level code when loading our LKM.

The more articulated actions for managing master page copies take place when a page is shared among processes—hence it is reachable via multiple address spaces. To deal with this scenario we included two different solutions depending on if the page is really shared—like pages that are mapped with the `MAP_SHARED` flag—or are implicitly shared—like for the case of Copy-on-Write (COW) pages of the address space.

For explicitly shared pages, we support the maintenance of our virtualized page permission scheme by relying on a combination of solutions. First, when a change in the shadow state machine for page permission (see Figure 4.2) is triggered by a specific process P, we identify all the address spaces that share the page exploiting the "page_info" data structure, from which we reach a list of page table pointers for all the processes that share the page. Then for all the associated page table entries that redirect to the shared page, we update the bits used for actual permissions—while leaving unmodified the bits used for managing our shadow state machine, which keeps virtualized page access permissions. Specifically, the virtual permissions associated with process P that is performing the update, expressed as the couple $< B_0, B_1 >$ (for W and X), are used to perform the AND with the corresponding ones in the entry of the page table of another process P' that shares the page to determine if the actual permission (W or X) that we grant to process P can be passed to process P'—process P' will anyhow retain its own virtual permissions. After, we update the corresponding entry of the page table of process P' depending on the outcome of this logical operation on the bits. We note that this activity is fundamental in order to avoid that a shared page that is accessible in W mode by some process and in X mode by another process could be used to bypass FlowScanner, leading to the execution of instructions that are arbitrarily written in the shared page by the process that has W permission—these instructions could in fact override the ill-op instructions posted by FlowScanner on the page.

As for implicitly shared pages (i.e. COW pages), FlowScanner allows different processes to use different actual copies of them, since it can generate different images of a same implicitly shared page given the opening of different code portions in the different process address spaces. (This may occur even if the different processes do not really write on the page). At the same time, the original COW nature is reflected by FlowScanner on the master page copy. More in detail, all processes containing the same COW page will share its master copy. When one of these processes attempts to overwrite the content of the

page, our system detects it, just exploiting the classical COW bit in the page tables. Subsequently, it creates a new master copy to accommodate the newly written contents of the page, thereby maintaining a consistent COW logic, similarly to how it operates in user space. At the same time, write operations on the COW page to open the different areas do not give rise to page faults since FlowScanner performs them still via the aforementioned mechanism of page remap at kernel level—which leads to accessing the page via other addresses and another page table entry.

4.1.5 Dynamic Check Engine

To facilitate a more flexible and easily updatable control system, we have incorporated a Dynamic Check Engine (DCE) into our architecture. When our Linux module is mounted, it exposes a kernel API, enabling other modules to add and remove callback functions. When the EM component tracks the execution of a new block, it passes the block content to the DCE component, which invokes the registered callback functions, passing the code block in input. Each function must return one of two outcomes: SAFE and UNSAFE indicating a safe block or a malicious payload, respectively.

The results are aggregated, based on the chosen configuration, and the aggregation determines if the payload is malicious. Currently, three configurations are available:

- All or Nothing: The block is considered malicious only if ALL callbacks return the UNSAFE result.
- Majority: The block is considered malicious if the majority of the callbacks return the UNSAFE result.
- At Least One: The block is considered malicious if at least one callback returns the UNSAFE result.

The selection of the configuration depends on how the individual check functions are implemented. These are choices essentially independent from the

FlowScanner architecture, which depend on policies that are selected for the management of the specific IT system hosting the applications that are subject to the analysis through FlowScanner. As an example, the check functions may work on determining whether the code block is identical to a specific signature, or is at some distance (below a threshold). In the later scenario, the operating modes based on majority or all-or-nothing can be suited for the identification of risks related to the program execution. At the same time, checking if the code block is identical to a specific signature leads the at-least-once policy to become relevant.

This system allows users to easily update the signatures for the engine, as the check functions are hosted in a separate module and are not tightly integrated with the main FlowScanner architecture.

5. System Resilience

The solutions presented in the previous chapter offer a good level of protection against a generic attacker, however, if we consider the environment of highly critical infrastructure, they are not sufficient. Malware detection tools like ours are effective at identifying known malicious software and suspicious patterns, but they often fall short when it comes to detecting insider threats—individuals within an organization who intentionally or unintentionally compromise security. These threats are difficult to identify because they involve legitimate users with authorized access, making their actions appear normal to automated detection systems.

In this environment, one of the most desirable protection guarantees is data integrity as unauthorized modifications or deletions of critical data—such as system logs and access records—can cause severe operational and security disruptions. Traditionally, these threats are mitigated using hardware-based solutions such as Write-Once Read-Many (WORM) storage. While effective, these approaches have notable drawbacks, including high deployment costs and irreversible consumption of storage blocks, which cannot be reused once written.

For these reasons, we developed a new solution that operates entirely at the software level while still ensuring data integrity across a commonly found range of threat environments. We call this solution VaultFS.

5.1 VaultFS

The problem of ensuring data integrity while maintaining availability/accessibility presents a complex challenge, particularly against powerful adversaries. The literature proposes various techniques based on different approaches and methods. Some solutions focus on detecting data corruption attacks (e.g., crypto-ransomware) and aim to block any actions that could cause further or irreversible damage to the file system as quickly as possible (see, e.g., [53]). These approaches primarily address "hot data", which remains accessible for both read and write (update) operations by active applications—an example are data kept by a database accessible by, e.g., a web server continuously updating the stored information. Other proposals rely on file system configuration, such as the setup and exploitation of file attributes (e.g., data immutability) [64], to ensure that stored files remain available in their original state and cannot be further modified. Furthermore, some approaches rely on specific file system instances that are accessible only under strict protection rules and in limited operating conditions [98]. These solutions are particularly suited for managing "cold data" such as medical imaging archives, legal/compliance records as well as backup data, which can be leveraged to restore applications to a correct operational state even after a data corruption attack, in alignment with a Recovery Point Objective (RPO) that reflects the application's criticality.

For this work, we specifically address the most general case of *continuous-time* cold data protection. This includes critical data that is continuously generated over time, such as access logs or streamed video surveillance frames, and requires an append-only semantic to ensure that past records remain immutable. To safeguard this data against corruption attacks while guaranteeing instantaneous accessibility—eliminating the need for any recovery procedure—we propose VaultFS, a Linux file system designed as a write-once register.

Approach. VaultFS is a software-only solution fundamentally distinct from, yet complementary to, existing methods in the literature. VaultFS enforces

sequential-only writes (no seek) within a single I/O session, preventing any modification, tampering, or encryption of already stored content. Once written, files remain fully accessible and can be exposed online (e.g. beyond a VPN), but become read-only, with no possibility of further write sessions, modification, or removal—not even by privileged (root) threads operating at the block device level.

Unlike hardware-bound Write-Once Read-Many (WORM) technologies (e.g., Microsoft's Project Silica [4] or SnapLock [22]), VaultFS operates entirely at the software level, supporting ordinary read/write devices like HDDs and SSDs. This flexibility enables integration with traditional file systems, enhancing redundancy and expanding data management options while ensuring compatibility with low-cost off-the-shelf devices and *true write-once semantics*, independent of hardware constraints. Overall, by offering WORM support purely at software level, VaultFS can definitely enlarge the possibility to protect data in differentiated environments while, at the same time, guaranteeing an easy and fast deploy of the data protection support without the need for strategies and decisions in terms of investments in hardware-level specialized storage technologies.

VaultFS is also fundamentally distinct from existing software-based immutability mechanisms. Unlike classical file system protections—such as setting the immutable "+i" flag via the `chattr` command—which can be reverted via privilege escalation, VaultFS permanently enforces append-only writes at the kernel level. As such, the write-once file-management semantic is guaranteed to be supported independently of the capabilities of running/subverted applications. Once a file is written, its content becomes read-only, with no possibility of modification, removal, or additional write sessions, even by privileged (root) threads operating at the block device level. Any attempt to operate at the block level of the device (or partition) that hosts VaultFS is intercepted and blocked along any time frame during which the file system is mounted. Furthermore, we configured a facility that enables the file system to be unmounted only at shutdown of the operating system kernel, prevent-

ing attackers from bypassing protection through privilege escalation after an unmount of the file system. All these features enable VaultFS to support arbitrary RPOs—namely, availability of data previously written at whichever point in time—in a manner fully independent of data backup software solutions which instead only enable data restore corresponding to an RPO that is explicitly linked to a backup operation. This is how VaultFS fills the gap for reaching continuous-time cold-data protection.

Technical challenges. For designing the system itself and ensuring security against file deletion, modification, or renaming, some core challenges have been addressed, which we list below:

1) A first key challenge resides in *guaranteeing controlled file retention*, i.e. allowing files to remain non-deletable and non-writable for a specified duration or indefinitely. Unlike baseline WORM devices (e.g., CD-Rs, DVD-Rs, Blu-ray discs, LTOs), which lack this flexibility, VaultFS enables configurable file retention at mount time, ensuring that files stay immutable for a predefined period before becoming eligible for modification or removal. Crucially, once set, the retention period cannot be altered, even under privilege escalation attempts, preserving the integrity of the write-once model while allowing for efficient storage management. This required the *custom design of a kernel-level trusted time management mechanism* to prevent an otherwise trivial attack—manipulating system time to bypass retention policies.

2) A second challenge relates to the design and implementation of mechanisms supporting an admission control policy suited for the write-once semantic, also considering that the device hosting a generic file system could be actually accessed using different entry points within the virtual file system architecture. This can take place particularly considering common system calls accessible by (effective)root-id threads. Overall, our design has required the identification of places of the Linux kernel requiring the inclusion of hooks for the appropriate run-time check and

management of the kernel side operations.

3) A third challenge in our solution resides in mitigating storage-exhaustion attacks. Indeed, VaultFS's write-once nature makes it susceptible to Denial-of-Service (DoS) attacks that could flood storage with immutable, useless files, preventing their removal until the retention period expires. As for this aspect, we implemented a runtime validation mechanism that restricts write operations to pre-approved (whitelisted) applications, configured at mount time. Any program not on the whitelist is blocked from writing to VaultFS. This approach ensures VaultFS remains viable for controlled environments where only designated applications (e.g., database backup tools like `mysqldump`) are permitted to write data.

5.1.1 Threat Model

VaultFS is capable of preventing unauthorized modification or deletion of previously written data under this threat model:

- The attacker can perform attempts via system calls, direct block device access, and memory-mapped I/O.
- The attacker can execute commands/software with root privileges.

At the same time, our trust assumptions are the following ones:

A) We assume a secure kernel and driver layer as part of the trusted computing base (TCB).

B) Our file system is mounted during the boot process before any attacker is able to execute tasks.

Overall, the above thread model represents the highest possible amount of power that one can give an attacker, (against a secure kernel), and, unlike many other security solutions, our system remains effective—in terms of write-once guarantees—under this threat.

As for the trust assumption in Point A, we recognize it is a simplifying assumption. In fact, many works have proposed automated kernel exploitation techniques [6], which may seem to contradict this point. However, most of this research focuses on privilege escalation, i.e., executing code with root-level privileges. As it will be clear, this alone is not sufficient against VaultFS since configuring the secure-boot prevents the possibility of mounting untrusted LKMs which could damage the operations of VaultFS. Furthermore, its LKM has no `cleanup_module` function—hence it cannot be unloaded unless when shutting down the kernel. As a result, any attacker targeting VaultFS would need a custom exploit specifically designed to disable the protections of our system. Also, VaultFS is designed to operate in high-security environments with access to critical data. In such an environment, the kernel hosting our system would be enhanced with all available security features offered by the community, such as Kernel CFI, Write-Once memory, and more [65, 35, 24, 85, 101], making it extremely complex to produce a reliable attack against VaultFS.

We also recall that the assumption in Point A is de-facto the basis for the operations of systems like Linux Security Module (LSM) [67], representing approaches orthogonal to VaulFS for deepening defenses into an operating system kernel. Additionally, in real-life contexts, such an assumption can be sustained when the kernel code is subject to formal verification of its correctness, an approach used scenarios of high-assurance security and reliability [90].

Still for Point A, VaultFS LKM can be configured to exploit the kprobe subsystem of the Linux kernel to install an entry handler to the `init_module` system call in order to prevent the loading of any other kernel module (either signed or not) after the VaultFS one has been loaded. This enables keeping the kernel configuration defined—despite actions by (effective)root-id threads, like attempts to mount modules for removing the aforementioned `kprobe`-based hook—even when no secure-boot support is adopted. Choosing this configuration can be another way to prevent kernel level subversions under attacks based on privilege escalation. Additionally, it could be combined with

some mechanism to maintain the hash of the VaultFS LKM at hardware level (e.g. via TPM) in order to verify that this same LKM is trusted when loading it.

As for the trust assumption in Point B, VaultFS activates its protections—including those operating at the block-device level—when it is mounted. Also, it does not support its unmount operation. To avoid attacks before the mount operation is executed, one possibility is to assign the responsibility of mounting our file system to an external network controller, which verifies the online status of our system before restoring external connections. Also, this point is automatically guaranteed in systems that use PXEs[1] [84], which are prevalent in modern data centers [15]. In such setups, the operating system image and its configuration are centrally maintained on a remote server and cannot be manipulated by an attacker to execute commands before file systems are mounted. Alternatively, the kernel, VaultFS, and related configuration files could be hosted on a traditional hardware WORM device, ensuring—still via network control—the integrity of the boot process until VaultFS is fully operational.

5.1.2 Baseline Concepts and Architecture

Working at the file system level in Linux requires considering the strict relation that various kernel-level subsystems have with the file system driver. In particular, it is typical that the parts of the driver of a specific file system (like the file operations it offers) exploit—or are exploited by—kernel-level services which are components of the more ample infrastructure of the Virtual File System (VFS).

The interactions with services that are "external" to the file system driver leads to the impossibility of guaranteeing the security levels we target by purely working inside the driver. Hence, our objective has been the one of constructing a comprehensive architecture surrounding the file system driver for guaranteeing that the security policies in place cannot be circumvented by malicious

[1]Preboot Execution Environment.

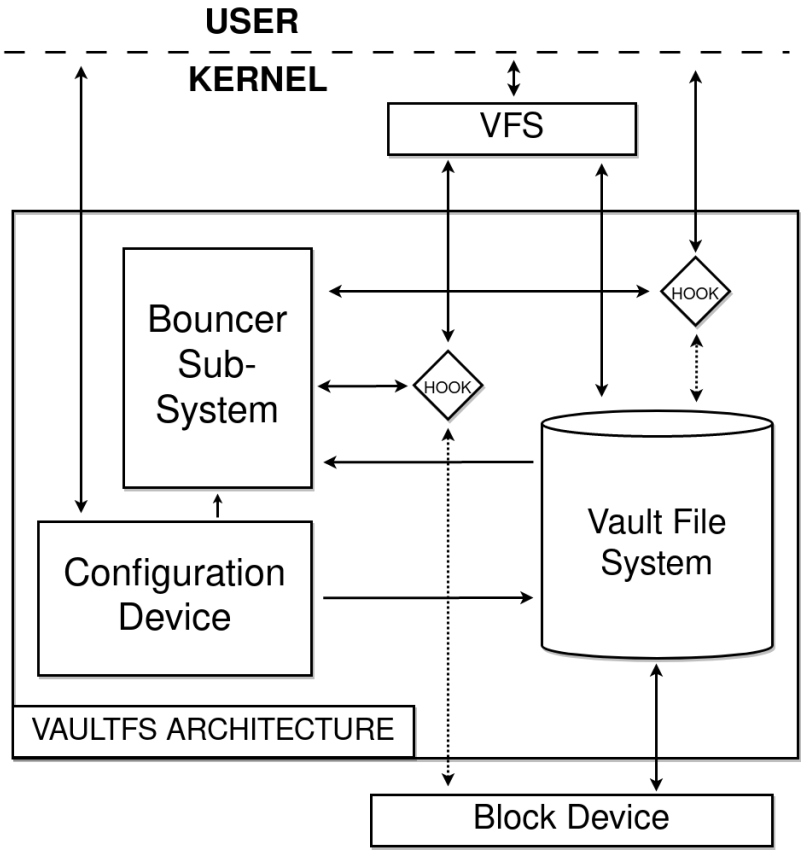

Figure 5.1: Architecture of VaultFS.

actors.

At a high level, the overall architecture of VaultFS can be schematized as shown in Figure 5.1. It includes three components, all of which can be added to the Linux kernel by relying on common APIs that Linux makes available for injecting any LKM. Hence, all our design is purely based on the LKM technology. The overview of these components is as follows:

- The *Vault File System* component is our file system driver. It takes care of managing files according to specific policies that are applied to any I/O session which can be ever opened by any process. In particular, it implements a fully new file system, inspired by Ext4, which only allows file writes along a single I/O session without overwriting—hence ensuring the write-once semantic. This feature fundamentally eliminates the possibility of data tampering while utilizing the file system, which is supported also via the embedding of admission control of operations working

at level of the file system driver. Nevertheless, despite these limitations, this file system driver is compatible with most backup software and is fully adequate for other scenarios involving cold data management. The decision to implement a new filesystem, rather than modifying an existing one, is motivated by the fact that our architecture offers specific functionalities in terms of data management which, for being guaranteed, would require massive modifications of any already existing file system. By designing a new filesystem, we are better able to focus directly on the target data management features we envisage. As a last note, the *Vault File System* implements admission control mostly inside the different file-system specific modules it provides, and uses the kprobe Linux support for nesting hooks just to manage memory-mapping operations involving the page-cache kernel subsystem.

- The *Bouncer Subsystem* provides its support for the admission control of operations supported externally to the *Vault File System* driver, still by relying on hook functions nested via the kprobe support. In particular, it prevents the unmounting of VaultFS, and denies write-access to the underlying block device.

  As shown in the picture, the *Bouncer Subsystem* also offers hooks for kernel level functions that are external to the VFS. In particular, it includes hooks for the management of the memory-mapping kernel-level function, which is used to setup the structure of the address space of the active processes. These hooks enable the *Bouncer Subsystem* to check the content of executable pages into an address space, in order to determine if an active process that tries to write on VaultFS can be considered as legitimate. In fact, the *Bouncer Subsystem* also replies to queries that the *Vault File System* driver can issue. Through these queries, *Vault File System* can decide to accept or reject a write operation on a file, which is fundamental for enabling the file system to mitigate the DoS attack.

- The *Configuration Device* enables configuring the VaultFS instance both

at mount time and, if enabled by an administrator, also at runtime. In the latter case, it can be exploited through VFS system calls in order to enable changes on the admission of operations performed by both the *Bouncer Subsystem* and the *Vault File System* components. However, it is important to remark that this device is not a security-critical part of our architecture, just because it can be configured to deny at runtime any configuration modification, even if the `ioctl()` calls for interacting with the device are issued by (effective-)root-id threads. This is the default restrictive setup, which enables to achieve the maximum security level, despite privilege escalation. This default can be modified at the LKM load time just to allow for greater flexibility, and for enabling the end-user to exploit VaultFS according to his own needs.

In the subsequent sections we present the details of each of the components being part of the VaultFS architecture.

5.1.3 The Vault File System

The centerpiece of our architecture is the *Vault File System*. It is a file system driver that we have developed in its entirety and is based on the layout and philosophy of Ext4. This file system driver is designed to host cold data and, as we pointed out before, it is founded on the write-once principle.

There are some fundamental properties that must be satisfied via operations on this file system, which we list below:

1) A free i-node which can be used for a regular file creation is set as busy-protected via an `open()` system call[2] that sets up an I/O session with write capability and that receives in input a file name representing a fully new hard link towards the i-node.

2) No additional I/O session with write capability can be opened on a busy-protected i-node related to a regular file, up to the end of the file-

[2]We use the `open()` system call as representative also for the `openat()` system call.

protection lifetime (see Section 5.1.6 for configurable exceptions). Hard links can still be set towards the busy-protected i-node.

3) Hard links towards a busy-protected i-node cannot be removed up to the end of the file-protection lifetime. Hence, the busy-protected i-node cannot become free up to the end of the file-protection lifetime.

4) Once a write on a file whose i-node is busy-protected has been executed the written data must never be altered for the whole file-protection lifetime. This holds also for the unique I/O session with write capability we can have on a busy-protected i-node.

5) The above properties imply that data-blocks indexed by a busy-protected i-node cannot be released (and the file data they keep cannot be over written) for the whole file-protection lifetime.

The above operations relate to regular files, but VaultFS also manages directories, in particular their removal, exploiting the busy-protected state of their i-nodes.

For supporting all the above mentioned operations, we exploited a state machine, schematized in Figure 5.2, whose current state is kept in a flag into the i-node. At file system formatting time, each i-node not used for the root directory is set as free. When an `open()` system call with write capability is executed using a file-name not corresponding to any existing hard link, the file system driver finds a free i-node and puts it into the busy-protected state, updating the i-node flag—see step 1 in Figure 5.2. This prevents the file system driver from opening any further session with write capability for the whole file-protection lifetime. This constraint applies unconditionally even if a process/thread runs with effective-root privileges or attempts to remount the file system with modified parameters (recall that, as we mentioned in Section 5.1.1, a VaultFS instance cannot be unmounted before the kernel shutdown).

Additional I/O sessions that only operate in read mode can be opened, either within or after the protection lifetime, and reading after a seek on any point of the file is supported in these I/O sessions.

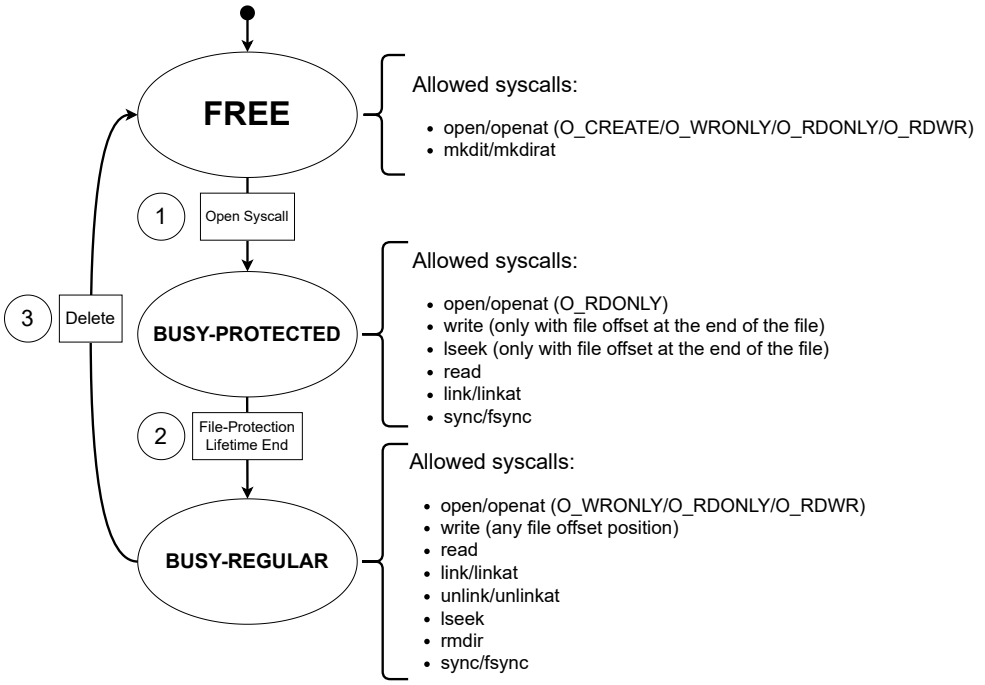

Figure 5.2: i-node state and operations—the execution of system calls not present in the table for managing files/directories is denied by default on VaultFS

As for files associated with directories (including the root directory), the *Vault File System* driver does not allow the removal of any hard link they keep up to the end of the corresponding file-protection lifetime (see point 3 in the above list). Hence any hard link to an i-node can be removed only after the i-node is no longer in the busy-protected state, i.e., it has been passed to the busy-regular state.

In order to control write operations when the i-node is in the busy-protected state (see points 4 and 5 in the above list), we leveraged the inherent behavior of conventional file systems. Upon the successful completion of a write operation, any file system driver will appropriately modify the size field within the i-node to account for the newly written data. We deem the write operation as committed at that particular moment and, consequently, regard all the data contained within the file up to that point as committed. This implies that any subsequent write can only be accepted if it does not modify any data existing

i-node state	Syscalls	Flags or Conditions	Allowed/Denied
Free	open/openat	O_CREAT, O_WRONLY, O_RDONLY, O_RDWR	Allowed
Free	mkdir/mkdirat	—	Allowed
Busy-protected	open/openat	O_RDONLY	Allowed
Busy-protected	open/openat	O_CREAT, O_WRONLY, O_RDWR	Denied
Busy-protected	write	file offset == size	Allowed
Busy-protected	write	file offset < size	Denied
Busy-protected	read	—	Allowed
Busy-protected	link/linkat	—	Allowed
Busy-protected	unlink/unlinkat	—	Denied
Busy-protected	rmdir	—	Denied
Busy-protected	lseek	file offset < size	Denied
Busy-protected	sync/fsync	—	Allowed
Busy-regular	open/openat	O_WRONLY, O_RDONLY, O_RDWR	Allowed
Busy-regular	write	any file offset	Allowed
Busy-regular	read	—	Allowed
Busy-regular	link/linkat	—	Allowed
Busy-regular	unlink/unlinkat	—	Allowed
Busy-regular	rmdir	—	Allowed
Busy-regular	lseek	any file offset	Allowed
Busy-regular	sync/fsync	—	Allowed

Table 5.1: i-node state and operations—the execution of system calls not present in the table for managing files/directories is denied by default on VaultFS

from the beginning to the current size of the file. This check can be easily implemented at the start of the “write" file operation inside the file-system driver. Furthermore, to adhere to the POSIX standard, we have modified the file operation “lseek" to prevent for any i-node that is busy-protected the movement of the file offset along the session with write capabilities to an area where write operations are not allowed.

However, we also need to consider that the file content in Linux is actually manipulated via the page-cache subsystem, which allows mapping page-cache pages into the address space of the applications. In particular this can occur when the `mmap()` system call is used for mapping a non-anonymous memory content into the address space; in this case one parameter that is needed for the system call is the file descriptor related to an open session on a file. This type of mapping is an additional channel an attacker might exploit to subvert the write-once semantic for files whose i-node is in the busy-protected state. In particular, even though there can be a single I/O session with write capability

on a busy-protected i-node (see point 2 in the above list), the corresponding file descriptor can be exploited in an attack (also according to privilege escalation) in order to memory-map the file and rewrite its content. In fact, such a mapping would lead the application software to gain direct control on the file content, with possibility of performing updates even in the scenario of a file that has been just opened (created) and populated via a unique session with write capability.

To avoid this scenario, our solution exploits the kprobe facility offered by Linux to install a hook on the `mmap()` system call. The hook checks if the file descriptor passed in input corresponds to an i-node in the busy-protected state. In the positive case, the mapping service returns with an error code if the corresponding I/O session has write capability, preventing the mapping operation. At the same time, the operation is still supported with no restriction for files whose i-node has passed to the busy-regular state.

Still in relation to the management of i-nodes, we decided not to support the creation of char/block devices via `mknod()` in VaultFS, except if a specific parameter is selected at mount time of our LKM (see Section 5.1.6). The reason for this choice is related to the fact that any i-node for a char/block device can be associated to any char/block-device driver offered by Linux. This driver is external to our *Vault File System* driver, hence our software cannot provide any support for ensuring that specific properties, e.g., in terms of security, are actually ensured for a busy-protected i-node created for such char/block device.

In Table 5.1 we schematize the behavior of the *Vault File System* driver in terms of system calls, and conditions driving (allowing/denying) their execution. As mentioned, any other system call that is not listed in the table is denied by default by this driver.

A further aspect of the architecture concerns the notion of the "lifetime" of i-node (or file) protection. This is the period during which an i-node must remain in the busy-protected state and cannot be reused for storing data from another file or for updating the data it already holds. Essentially, this lifetime

represents the duration over which the write-once property is guaranteed for files in the file system.

This lifetime can be ideally set as unlimited, which means that a file hosted by a mounted instance of the file system can be subject to no removal and no rewrite at any point in the future. This is the scenario where the state machine depicted in Figure 5.2 boils down to a two-state configuration since the busy-regular state of the i-node can never be reached along time (in particular via step 2 in Figure 5.2). However, if we consider the case of exploitation of our file system for keeping backup data (e.g. database or VM backups to be kept accessible, although protected, for dealing with any critical scenario), the impossibility of adopting a finite lifetime of the protection could make the cleanup of no longer needed backups and data challenging.

To cope with this aspect in VaultFS the Protection Lifetime (PL) can be configured at the file system mount and represents a time interval, starting from the file creation, after which the file is considered obsolete for what concerns write-once insurance, and, consequently, all (over)write and delete limitations (including the delete of hard-links) are lifted. After this PL, the file i-node passes to the busy-regular state, and is managed according to the conventional (regular) operations any common file system, like Ext4, supports. This configuration offers a more straightforward maintenance process and, in general, increases flexibility, as the PL can still be set to an infinite value for specific cold-data management scenarios—like the one where the file system is used for files representing MRI data in healthcare systems.

The implementation of the PL feature requires in its turn some careful considerations in order not to become a critical point for security. In particular, in order to determine whether to lift the restrictions or maintain an i-node in the busy-protected state, it is crucial to possess a reliable method for measuring the passage of time within the system. However, the current Linux kernel APIs do not provide a suitable solution for this task. The real-time ktime accessors return a value that can be modified by both users and NTP servers [66], while the values provided by the boot time and monotonic clock are reset during

each system shutdown. It is possible to intercept the shutdown process and update an internal counter to track the actual elapsed time, but this method would prove ineffective in cases of forced power-off such as power outages.

To completely resolve this issue, we have included in our module a kernel daemon in the form of a workqueue item, which is triggered at regular intervals. The only task of this daemon is to update an internal time counter known as the “file system time" (FST), which resides within the superblock of the file system instance. FST represents the actual time that has elapsed since that specific file system instance was mounted for the first time. We note how FST is active only along periods where VaultFS is mounted, since its superblock is reachable via the function associated with our work-queue item only in this scenario. However, this is perfectly aligned with the objective of VaultFS, which has been thought as a storage system to be constantly kept mounted and active (for enabling the access to the recorded files), while still being protected against attacks like, e.g., ransomware, even under privilege escalation.

In the current setup FST can operate at the granularity of an individual second, which is the update interval of the daemon, even though we suggest setting it to work with granularity of the order of 10 seconds, so as to actually achieve no interference at all with the operations of other workqueue-items the Linux kernel can setup. Also, given that this time measurement is specifically utilized for checking the file-protection lifetime, and for passing an i-node from the busy-protected to the busy-regular state, which typically spans periods like weeks, months or even years, according to classical data-maintenance, higher precision (less than 1 second granularity of the timer) is deemed unnecessary. With this architecture, our file system ensures resilience to power-off events and system crashes as the value of FST is periodically saved to the disk partition that hosts VaultFS. Even in the scenario where a shutdown occurs just before the daemon’s time update, the discrepancy in the FST value would be less than the setup period (e.g. 10 seconds). Considering the context, we believe this error is negligible. Furthermore, even in such a scenario, FST would be lower than its theoretical correct value, resulting in a prolonged lifetime of file

protection. In other words, no violation of the selected PL can ever happen considering the event of too early elimination of data from the file system.

Still considering this point, regulations like GDPR [33] indicate that (personal) data must be kept no longer than necessary for the purposes for which it was collected. In this case the PL can determine when to remove information. Using VaultFS in the system design would require determining the maximum delay of the actual data-remove operation with respect to the PL expiration, in order to just setup the granularity of FST considering, as discussed above, that it is a lower bound on the actual data PL.

In Figure 5.3, we illustrate the usage of FST in two specific scenarios. Firstly, during the file creation process, VaultFS reads the current value of the internal time counter (FST) and stores it within the corresponding i-node of the file—see step 1 in Figure 5.3. Secondly, when a delete request is received, VaultFS retrieves the current value of FST, subtracts the saved internal creation time obtained from the i-node, and compares it against the PL value set by the system administrator. If the calculated time difference exceeds the PL, the i-node is passed to the busy-regular state and the delete operation is permitted; otherwise, it is denied—see step 2 in Figure 5.3.

In Chapter 6, we will demonstrate that the VaultFS architecture does not conflict with existing software tools and will not impede both the backup process, via classical backup programs, and the storage of data coming from video surveillance applications, which are exploited in this article as a use case in relation to cold-data management.

In any case, as we will show in Section 5.1.6, the VaultFS LKM also offers (1) a flexible mount scheme where options can be configured and (2) the optional support for runtime reconfiguration of the file-system operations, which can be exploited in scenarios where extremely high flexibility is requested and specific trade-offs between security and the capability of supporting tasks is the objective for its usage.

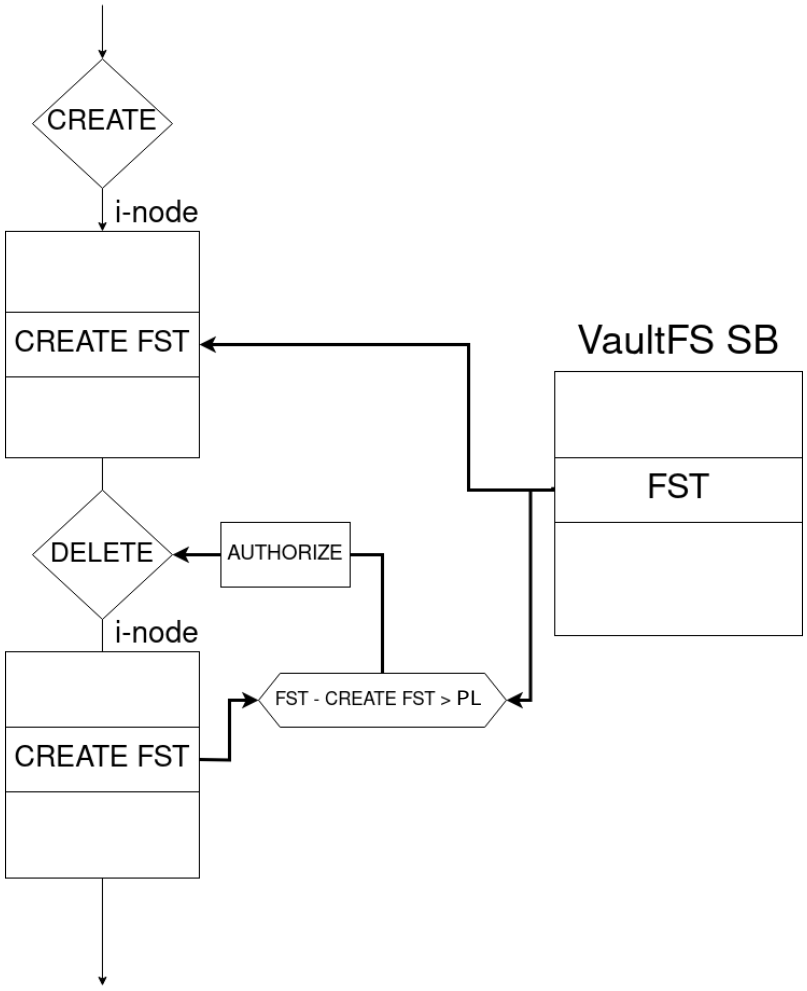

Figure 5.3: Create and delete operations with FST - it shows how the FST value is exploited when either creating a file (step noted as 1) or trying to delete it (step noted as 2).

5.1.4 The Bouncer Subsystem

The *Bouncer Subsystem* we included in our architecture has two main goals: firstly, to ensure that the only possible way to access data on the file system is through the *Vault File System* driver, and secondly, to ensure the up-time of VaultFS.

The first objective is self-evident: VaultFS requires exclusive access to the storage device to protect it with its restrictive write-once permissions, at least for files whose i-node is currently busy-protected. However, this is not typically the case in most modern VFS architectures, as access to the storage device is permitted not only from the file system driver, but also directly from the

block device beneath it. To mitigate this vulnerability, the *Bouncer Subsystem* intercepts all mount events and records the major/minor numbers of all block devices hosting a VaultFS instance. Subsequently, as shown in Listing 5.1, when a user attempts to access a path using the `open()` system call, the *Bouncer Subsystem* intercepts the call, recovers the id of the block device hosting the file which is being opened (see line 2), and verifies whether it is one of the previously recorded major/minor numbers (see lines 5 and 6), and if so, the operation is denied (see line 8).

```
1  //check if we are accessing a device directly
2  bdev = name_to_dev_t(path);
3  if (bdev) {
4   //check if device is protected
5   list_for_each_entry(info, &vaultfs_instances, n) {
6    if (info->bdev_id == bdev) {
7     //it is protected - fail
8     res->pass = 0;
9     break;
10   }
11  }
12 }
```

Listing 5.1: Limiting block device access

Furthermore, keeping VaultFS active means not only guaranteeing the write-once semantic, but also that the file system is kept mounted, in order to protect the access to the block device according to what explained before (see the Listing 5.1). To address this issue, the *Bouncer Subsystem* installs a kernel probe on the `umount()` system call, as shown in Listing 5.2. This probe intercepts the call and compares the target path with all active VaultFS instances (see lines 5 and 6). If a match is found, the probe checks whether unmount protection is enabled (see line 8). Since this protection can be disabled by a VaultFS administrator, the outcome depends on its status: if active, the unmount request is blocked (see line 10); if not, the unmount proceeds and the corresponding VaultFS metadata is removed.

As a final note, all security metadata for the subsystem is generated at mount time from the file system instances and stored in read-only volatile memory, providing an additional layer of protection and preventing corruption.

```
1  //pass by deafult
2  res->pass = 1;
3
4  //check if path is protected
5  list_for_each_entry(info, &vaultfs_instances, n) {
6      if (strcmp(info->mount_path, path) == 0) {
7       //is is protected - is the lock active?
8       if (info->umount_lock) {
9        //it is active - fail
10       res->pass = 0;
11      } else {
12       //it is not - clean up the list - pass
13       list_del(&info->node);
14      }
15      break;
16     }
17 }
```

Listing 5.2: Blocking umount

5.1.5 DoS Mitigation

An additional mitigation we included in the design of VaultFS tackles DoS attacks. In particular, ensuring that, according to some capacity plan, it still has space to store data, could be central in automatic backup or data store procedures. However, this represents a more problem since, due to the pseudo-permanent nature of the files in VaultFS, an attacker could employ a DoS attack by filling up the hard drive with useless yet still non-erasable files, rendering the system unavailable. The same is true for hard-links, kept into the directory files. To address this issue, we have incorporated a whitelist of programs that can be added during the file system mount by a system administrator. Any program not included in the whitelist will be prohibited

from writing to the VaultFS instance, thereby hardening the possibility of a DoS attack. The discovery on the possibility to write data is done by the *Vault File System* driver via queries issued to the *Bouncer Subsystem*. These queries are issued when the `open()` and `write()` system calls are handled, as well as for the handling of the `link()` system call, which manages the creation of hard-links.

The creation of the whitelist and the authentication process can be schematized as shown in Figure 5.4, and work as follows:

1. During the hard drive formatting a list of approved programs and libraries can be included. This list is stored in the file system superblock.

2. While mounting VaultFS, our software reads the list, identifies the files associated with the approved programs/libraries, and generates an hash for each file, in particular for executable sections. These hashes are then stored in memory along with their corresponding file absolute paths.

3. While VaultFS is mounted, whenever a new process maps an executable memory area associated with a file, we again intercept the mapping. If the path of the allocated area matches one of the authorized paths, we compute the hash of the area and compare it to the corresponding stored hash. If there is a match, the process is considered authorized. If there is no match, the process is permanently banned from accessing VaultFS in write mode, and any future memory mapping made by that process are not monitored.

The central aspect of this architecture revolves around the computation of hashes. The efficiency of this calculation is crucial as it is performed synchronously during memory mapping. Simultaneously, we have to ensure the security of the hash to prevent malicious attackers from easily bypassing the check. To strike a balance between speed and security, we have implemented the following rules:

- Any process with an executable `vm_area` entry whose file name is not

present in the whitelist is automatically banned without hash computation. This measure can effectively prevent the majority of memory mapping operations from triggering hash calculation in case a large volume of executions of non-white-listed applications takes place.

- At file system mount time, for each whitelisted program, we compute hashes related to each of the different executable sections used by the program (including libraries). Each hash is computed after the selection of a specific percentage of cover of the section, which we refer to as area. At the same time, all the hashes associated with a section are required to globally cover all the section content.

  The size and position of these percentages of coverage are determined using a random number based on a random seed determined during the file system mount process, which is specific to each VaultFS instance. This means that the stored hashes change every time when remounting the VaultFS instance (e.g. at a subsequent operating system boot). When in a process this same file (program) is executed, and the mapping of its sections takes place, we randomly select what area(s) of the section to hash and check against the originally computed hash values. To ensure security, the areas selected for the hash of each executed instance of a same program are as well determined randomly. This approach guarantees that an attacker cannot determine which specific area is being used for the hash and consequently cannot construct a malicious authorized binary. At the same time, our system hashes only one or a few randomly selected areas that belong to a section at each program execution (rather than all the program sections), reducing the cost of this operation.

This solution is based on the determination of the hashes of sets of pages as computed at the mount time of VaultFS. Hence, we prevent the inclusion in the white list of any program that includes memory zones (sets of pages) that are marked as write-exe, since their content can be changed at runtime.

To fully understand this process, let us walk through an example using the

`mysqldump` program:

1. **File system formatting:** during formatting, the administrator registers `mysqldump` as an authorized program. The VaultFS `mkfs` utility inserts the absolute path-name of this program (e.g. `/usr/bin/mysqldump`) into the file-system superblock, together with the path-name of the required libraries
   (e.g. `/lib/x86_64-linux-gnu/libpthread.so.0`).

2. **File system mounting:** when the file system is mounted, for each of the path-names kept in the superblock, a set of N hashes $\{H_0, \ldots, H_{N-1}\}$ is computed, where each hash represents a random subset (hence a given percentage) of the program's executable region, such as the region associated with the `/usr/bin/mysqldump` file. Such random subset corresponds to the area of a section we mentioned before. These hashes are then stored into read only volatile memory, enabling fast access whenever verification is required.

3. **Program execution:** after the file system is mounted, if the process running `mysqldump` attempts to write to VaultFS, the system iterates through its `vm_area` entries with execute permissions. For each area, for example the one associated with `/usr/bin/mysqldump`, one entry H_x of the predefined hashes $\{H_0, ..., H_{N-1}\}$ is selected and the corresponding data in `/usr/bin/mysqldump` are used for recomputing that specific hash value and comparing it with the original H_x value. If the comparison of the hashes fails for at least one of the executable `vm_area` entries, the write operation is denied. Additionally, writes are also denied if a memory area with `WX` (write + execute) permissions is detected, or if an executable `vm_area` corresponds to a name for which no hash exists in VaultFS.

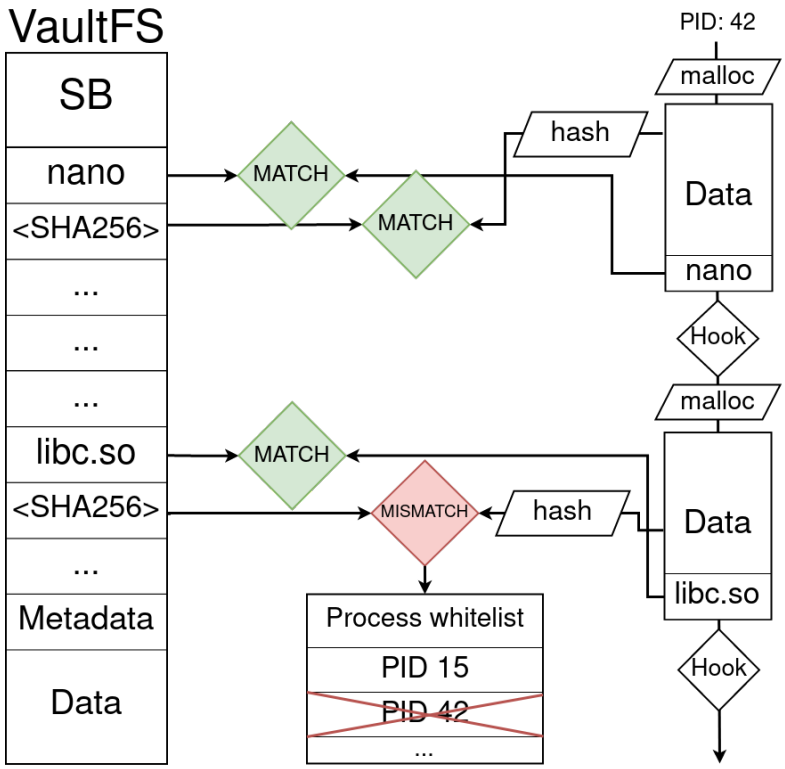

Figure 5.4: Whitelist example - the process with PID 42 is excluded from the white list since its shows a mapped memory region with hash not corresponding to the reference one kept in the VaultFS superblock

5.1.6 Configuration Device

To facilitate integration of VaultFS with existing infrastructures, we have incorporated a variety of configuration options that enable system administrators to customize the file system functionalities to their specific needs. Each option can be selected during mount by configuring the appropriate value in the file system superblock, with the configuration being specific to each instance, thereby allowing for different configurations on different partitions of the hard drives. The currently allowed settings include:

- *PL*: this option defines the file-protection lifetime. If the option is not enabled at the file system mount, the value *"infinite"* is used as the default.

- *Append reopen*: this option determines if a file having its i-node in the busy-protected state, which is not currently opened in write mode, can

be reopened in write mode for appending new data at its end. This parameter can be useful when VaultFS is exploited for securely keeping log files, which can be reopened in append mode by the applications. With VaultFS these files can still be kept on-line with no risk of tampering of their content via attacks, even under privilege escalation. This option is disabled by default.

- *Programs whitelist activation*: this setting activates the program whitelist as mentioned in Section 5.1.4. By default this option is disabled.

- *Mount unlock*: this option enables the system to be unmounted without shutting down the operating system, using a per-instance password. This option is disabled by default.

- *Char/block devices creation:* this option determines if the creation of char/block devices in directories of VaultFS is allowed. This option can be also configured in order to enable the creation after passing through a query to the *Bouncer Subsystem* in order to detect if the process that is calling the creation is whitelisted. By default this option is disabled.

- *Communication device*: this option determines if this instance of VaultFS needs to accept configuration directives at runtime, with the possibility of dynamically reconfiguring any of the above listed options. Also this option is disabled by default.

The default setup for all the listed options is summarized in Table 5.2. Such default setup is a way for making VaultFS work according to a pure mandatory protection approach preconfigured and immutable by any system user.

Alternatively, the *communication device* can be configured as enabled. In such a case, the LKM of VaultFS includes a dedicated `ioctl()` device that can directly modify the superblock of the file system instance to change its behavior, after receiving directives from authorized users. An authorized user is identified with a per-VaultFS-instance password, chosen during mount, which cannot be changed and is securely stored inside the superblock. Using this

Option	**Default setup**
PL	enabled
Append reopen	disabled
Program whitelist activation	disabled
Char/block device creation	disabled
Communication device	disabled

Table 5.2: Default setup of the configuration options of VaultFS

device, a system administrator can change all the settings that were decided during mount. It is important to note that all configuration data is stored inside the i-node of each file. This design guarantees that configuration changes are not retroactive: files will always maintain the configuration that was active at the time of their creation. As a result, files with different configurations can coexist within the same VaultFS instance, and modifying the configuration will not reduce the security of files that were previously written.

6. Experimental Evaluation

In this section, we provide an experimental assessment of all proposed solutions, the machines utilized for the experiments have the following specifications:

- **For VaultFS** all the tests we describe have been carried out on a standalone desktop computer equipped with an Intel i7 processor running at 3.1 GHz and 12 GB of RAM. The instances of VaultFS, Ext4 and Ntfs, which we compared with each other for what concerns performance, were all kept on identical 32 GB USB drives. The LKM module supporting VaultFS has been designed to ensure compatibility with a recent and still supported Long-Term Support (LTS) version of Ubuntu, number 22.04, which incorporates kernel version 5.15. Also, we configured Ubuntu with an Ext4 root file system.
- **For JITScanner and FlowScanner** all tests have been run on a ProxMox Virtual Machine with 4 virtual CPUs and 32 Gb of RAM, running Debian 12 with Linux Kernel 6.8 as a guest operating system.

6.1 Compatibility

Compatibility is a critical requirement for security solutions, as policies that cannot be easily integrated into existing infrastructures are unlikely to be implemented, and designs that impose significant user restrictions are often disabled to reduce frustration. For this reason, the solutions presented in this study have been designed with compatibility in mind. JITScanner and FlowScanner are completely transparent to end users and therefore pose no barriers

to adoption. In contrast, VaultFS is not fully transparent, as it enforces strict file management policies. For this reason, we conducted an experimental study to ensure that our solution is compatible with common use cases; the results are presented in this section.

6.1.1 VaultFS

VaultFS has been specifically designed for storing and managing cold data. Considering this target, we concentrate on assessing the compatibility of VaultFS with established software in the backup and video surveillance domains. In any case we recall again the ample set of use cases that VaultFS has, like for example the maintenance of medical test results. Applications like the latter mentioned one have not been considered in this study simply because of the unavailability of proprietary software, the use of which would have been necessary for the experimental phase.

Backup

In order to evaluate the compatibility of VaultFS with backup tools, we selected widely used software solutions for both file system and database backups. These tools have been used to perform a backup operation from an Ext4 file system to a VaultFS instance. We configured VaultFS according to both the default setup (denoted as "Default Protection") indicated in Table 5.2, and with the "Append Reopen" option activated. In any case, the life time of the file protection has been set to one year. We recall that under both the used setups of VaultFS, there is no possibility to rewrite any single byte of a file (or a directory file) for the whole protection life time, even when threads run with (effective)root-id capabilities.

The findings of this experiment are presented in Table 6.1. The results clearly indicate that VaultFS is compatible with all the tested backup solutions, across both the tested configurations, without requiring any modification to the default backup-tool options. However, it should be noted that for the `unison` and `rsync` tools, an additional flag is necessary to prevent folder renaming

	Append Reopen	Default Protection	Details
GUI Copy Paste	✓	✓	None
rsync*	✓	✓	Requires flag to prevent folder renaming
flexbackup	✓	✓	None
unison*	✓	✓	Requires flag to prevent folder renaming
mysqldump	✓	✓	None
Deja-dup	✓	✓	None
Kbackup	✓	✓	None

Table 6.1: Compatibility with backup tools.

during the copy process, as this operation requires the elimination of a hard link, which is not permitted by VaultFS for the whole protection lifetime of any file (including directory files).

Video Surveillance

Similarly to the backup compatibility experiment, we conducted a test with video surveillance software. We selected multiple commonly used software solutions in this field, which differ in terms of how the video data is stored and the configuration settings associated with their setup. For this experiment, we configured each software to save the video or photo data captured by an USB webcam onto a VaultFS instance. At the same time, we made each software tool configuration files (or software specific files) reside on the root file system instance (as we mentioned this is an instance of Ext4) mounted by Linux, as it typically occurs for most of the applications. Also in this case we executed the tests by considering both the "Default Protection" offered by VaultFS and the activation of the "Append Reopen" option. In both cases, we configured one year life time for the file protection.

	Append Reopen	Default Protection	Details
ZoneMinder	✓	✓	None
Webcamoid	✓	✓	None
ivideon	✓	✗	Uses multiple I/O sessions to save the video files
xeoma	✓	✓	None

Table 6.2: Compatibility with video surveillance tools.

The results of these tests are summarized in Table 6.2. As we can see, almost all the software tools have been successfully used with VaultFS in both its

configurations ("Default Protection" and "Append Reopen"). The unique tool that required the "Append Reopen" setup for being correctly used is `ivideon`.

6.2 Performance

Ensuring an acceptable level of performance is crucial for all security solutions. In this section, we present data assessing the performance impact of all the proposed solutions, taking into account their various components. Our objective is to demonstrate that none of the systems introduces substantial or unacceptable performance overhead.

6.2.1 VaultFS

File system statistics

VaultFS, despite being inspired by Ext4 file system, is still a fully new file system, because of its peculiarities in the management of i-nodes and file data. Therefore, as an initial test, we aimed to show that its performance is comparable to other commonly used file systems in the Linux world. To assess this, we performed a file copy from/to each of the file systems we are comparing, namely Ext4, Ntfs[1] and VaultFS. The test is conducted twice: first with all file systems mounted on a 32 GB USB hard drive, and then with them mounted on a 2 TB SSD. The objective of these tests has been measuring the time taken to complete the file copy operation for files of various sizes, ranging from megabytes to gigabytes. Each value we report is the average over 100 samples, and we also report the standard deviation over the gathered samples.

The results of this test are presented in Figure 6.1 and 6.2 and summarized in Tables 6.3 and 6.4. The results show that VaultFS outperforms NTFS and has only a small performance drop compared to ext4. This difference becomes more noticeable with larger files on the USB drive and with very small files on the SSD, with a 16% overhead being the worst case. Considering that

[1]We used the Linux Kernel implementation of the Ntfs file system, contained in the */fs/ntfs* folder of the source code.

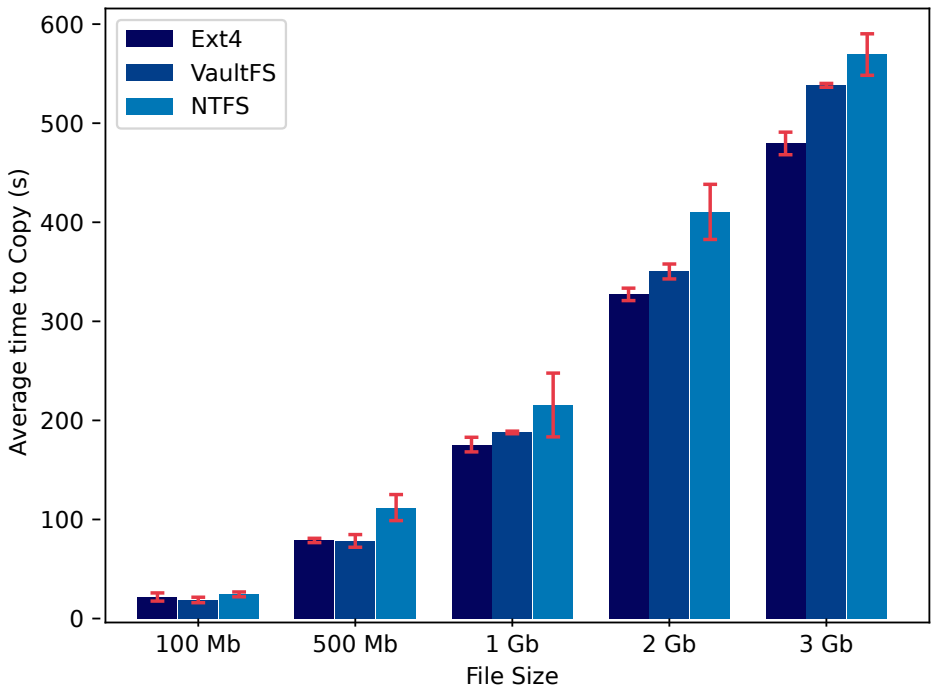

Figure 6.1: Average and Standard Deviation of the time to copy a file from an Ext4 instance to a Ext4/VaultFS/NTFS instance mounted on an USB Drive.

this result is obtained relying on the cp program, which essentially performs a continuous iteration of read/write operations from/to source/destination files, this test represents a kind of worst case scenario for the impact of VaultFS on threads' activities. In fact, the thread that is executing the cp program does not experience any significant latency caused by software components that are outside the VFS architecture of Linux, which would have reduced the relative cost/overhead of file system operations[2]. For this reason, we argue that if we were to measure other interactive applications, such as a web server or a database engine, the performance gap between the file systems would likely be smaller. In any case, VaultFS can be subject to fine tuning and additional software optimization along time (like pre-reserving of blocks for new file data), specifically linked to performance improvements.

[2]The thread running the cp command can experience a delay related to the materialization in RAM of the virtual pages it is using for hosting data read from the source file, but this is somehow negligible considering the iterative structure of the program.

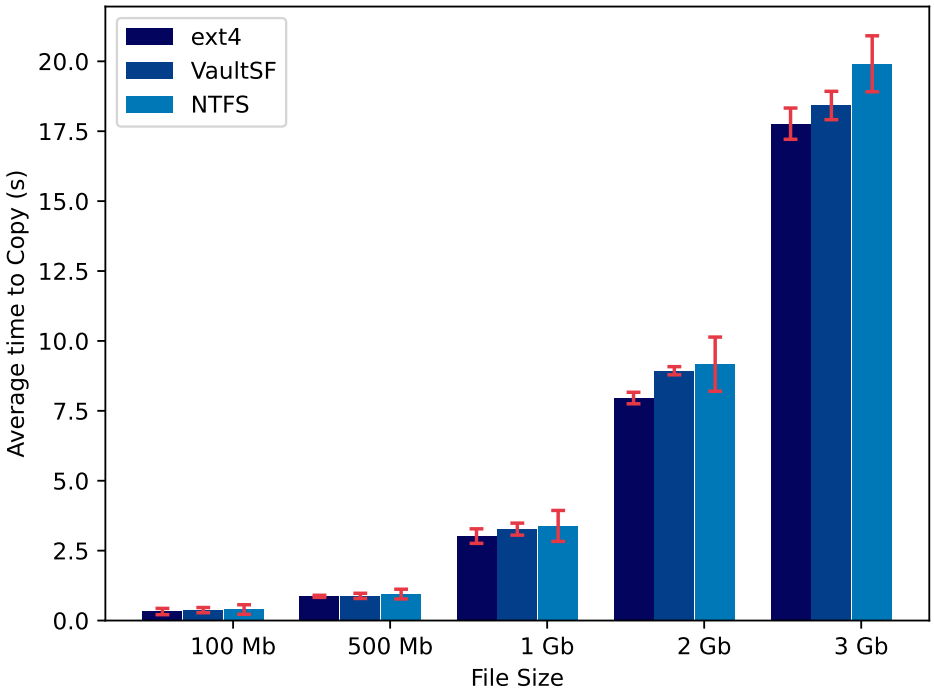

Figure 6.2: Average and Standard Deviation of the time to copy a file from an Ext4 instance to a Ext4/VaultFS/NTFS instance mounted on an SSD Drive.

Open kernel-probe slowdown

As we have explained, the *Bouncer Subsystem* requires a kernel probe on the `open()` system call, which is frequently invoked during normal system usage. Hence, it is crucial to ensure that our probe does not significantly impact the execution time of this system call. To assess this, we conducted multiple open operations on an Ext4 file system instance, still hosted on the aforementioned 32GB USB hard drive, measuring the time taken to complete them with and without the VaultFS LKM loaded.

The results, categorized by path depth, are illustrated in Figure 6.3. The value zero for path dept indicates that the `open()` system call opens a file in the current Process Working Directory (PWD). Each reported value is still the average of 100 runs. We would like to note that the values provided were determined under the assumption that the path being opened is absent from the page-cache. To ensure this condition, we unmounted the file system after

Disk Type	Size	Ext4	NTFS	VaultFS
USB	100 MB	21.74	24.40	18.74
USB	500 MB	78.81	112.00	78.35
USB	1 GB	175.59	215.56	187.84
USB	2 GB	327.21	410.47	350.33
USB	3 GB	479.56	569.30	538.21
SSD	100 MB	0.322	0.394	0.373
SSD	500 MB	0.866	0.946	0.882
SSD	1 GB	3.020	3.384	3.269
SSD	2 GB	7.958	9.170	8.934
SSD	3 GB	17.770	19.912	18.420

Table 6.3: Average time to copy a file from an Ext4 instance to a Ext4/VaultFS/NTFS instance.

Disk Type	Size	Ext4	NTFS	VaultFS
USB	100 Mb	1.0	1.12	0.86
USB	500 Mb	1.0	1.42	0.99
USB	1 Gb	1.0	1.23	1.07
USB	2 Gb	1.0	1.25	1.07
USB	3 Gb	1.0	1.19	1.12
SSD	100 MB	1.00	1.22	1.16
SSD	500 MB	1.00	1.09	1.02
SSD	1 GB	1.00	1.12	1.08
SSD	2 GB	1.00	1.15	1.12
SSD	3 GB	1.00	1.12	1.04

Table 6.4: Time overhead to copy a file from an Ext4 instance to a Ext4/VaultFS/NTFS instance (Ext4 baseline).

each measurement. Notably, there is no discernible slowdown caused by our module, indicating that the execution time of the `open()` system call remains largely unaffected when it becomes blocking since there is at least one element of the path (e,g. the specific file name) whose i-node needs to be read from the USB hard drive.

In any case, for completeness we also report in Figure 6.4 data related to the opposite scenario where all the items in the path are already kept in the page-cache. This time the actual execution of the thread opening the target file is essentially non-blocking, and we expect the maximum impact of the kernel probe we installed on the `open()` system call. From the data we observe a large variance, but it still appears that the impact of VaultFS is negligible.

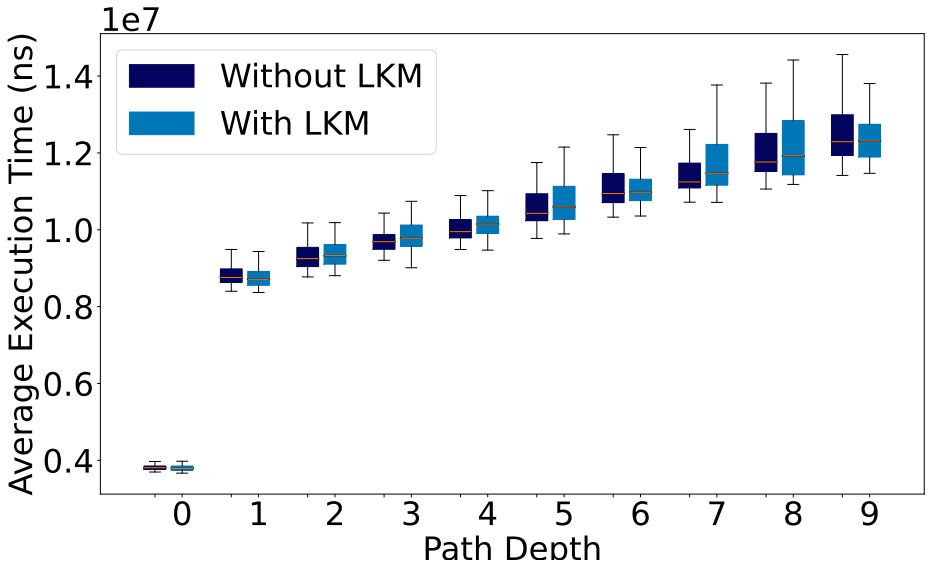

Figure 6.3: Box plot of the time for opening a non-cached file on an Ext4 instance with and without the LKM supporting VaultFS.

Process whitelist slowdown

The whitelist facility likely introduces the most significant performance overhead in VaultFS, as it adds operations during every memory mapping event while managing the address space of a process. To assess the acceptability of this performance overhead, we conducted several tests. The objective was to measure the execution time of multiple command line utilities in three scenarios:

- normal environment (without VaultFS);
- with VaultFS mounted and without the command line software in the whitelist; and
- with VaultFS mounted and with the command line software in the whitelist.

Furthermore, as explained in Section 5.1.4, we have configured VaultFS to hash areas representing different percentages of the process address space (i.e. of its executable sections) and measured how this affected performance. It is important to note that larger coverage of hashes checked represents a stronger support to security against DoS, but also introduces more performance over-

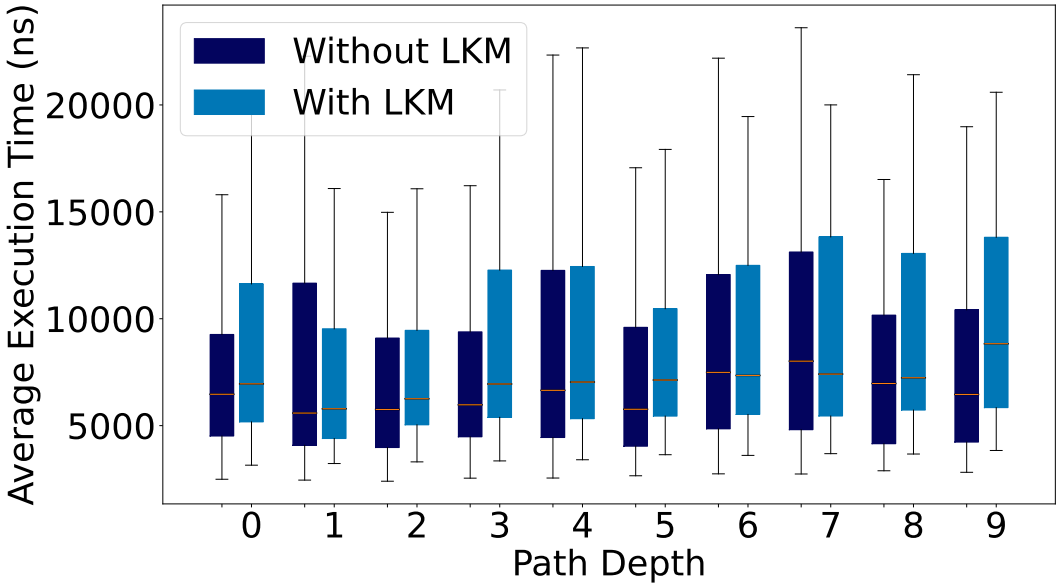

Figure 6.4: Box plot of the time for opening a file with all path structures in the page-cache on an Ext4 instance with and without the LKM supporting VaultFS.

	rsync (1Kb)	cat (1Kb)	cat (10Kb)	cat (100Kb)	zip (1Kb)	zip (10Kb)	ls
Not in Whitelist	0%	1,4%	1,5%	0%	0%	0,7%	4%
10% Hash	0%	15%	2,9%	0%	0,4%	4,7%	7,1%
25% Hash	7,4%	173%	87,9%	12,5%	7,4%	1,3%	131%
50% Hash	12,7%	344%	155%	23,8%	8,9%	1,4%	221%
100% Hash	23,5%	483%	227%	31,7%	13,5%	2%	285%

Table 6.5: Slowdown for common software.

head. The results are depicted in Figures 6.5 and 6.6, and the slowdown values are reported in Table 6.5. We have noted that when performing the hashes of the portions of each executable section, the number of pages involved in the hash operation has changed between 2 and 10 depending on the size of the section and the percentage of section-coverage selected when computing each hash.

We have categorized the values based on the average execution time, considering any program with an average execution time of less than 0.01 seconds in the normal environment as a short-lived application. We can observe the following trends:

- When the application is not in the whitelist, the slowdown is negligible ($< 1\%$).

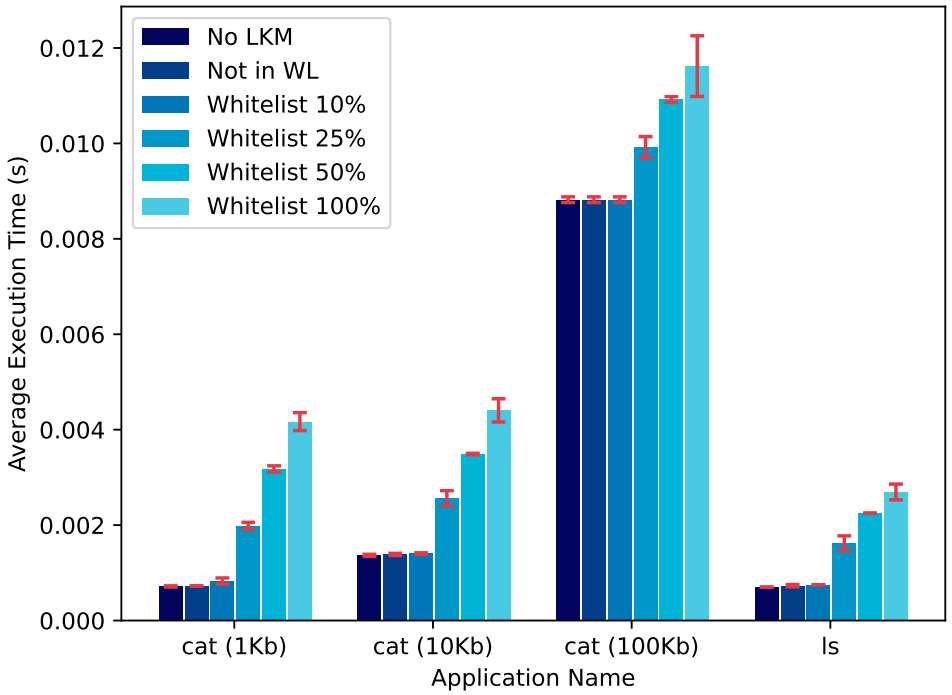

Figure 6.5: Average execution time for short-lived command line software — less than 0.01 seconds per execution in the normal environment (average and standard deviation over 2000 executions).

- When hashing areas representing the 10% of the section they belong, we observe a low performance overhead (< 16% slowdown) for all applications.
- When hashing areas representing the 100% of the section they belong, long-lived applications do not experience any significant slowdown (< 24%).

Given the very low delay representing the barrier between short and long lived applications, the data show how our solution is essentially non-intrusive under all the scenarios where a non-minimal amount of work is requested for managing/storing data. At the same time, in scenarios where the required amount of work is minimal, the applications under consideration are extremely short-lived, typically completing within a few milliseconds. At this timescale, in practice, the execution completes so quickly that the additional cost becomes

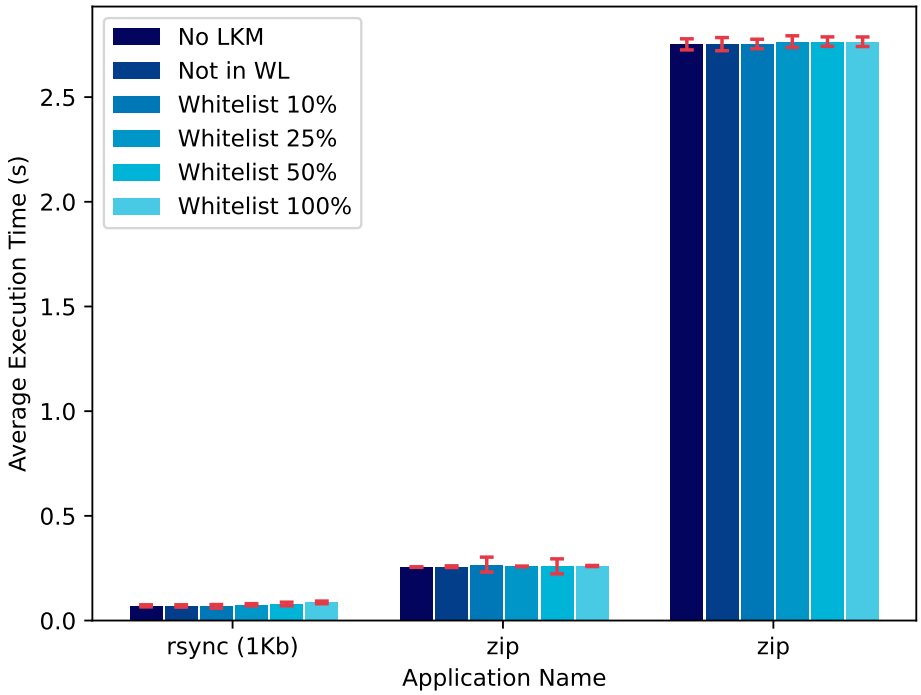

Figure 6.6: Average execution time for long-lived command line software — more than 0.01 seconds per execution in the normal environment (average and standard deviation over 500 executions).

effectively negligible from the perspective of the user or system throughput. Additionally, if the performance overhead for such short-lived applications is still a concern, VaultFS can be configured to apply hashing only to smaller portions of the address space, thereby reducing the associated cost even further while maintaining the desired level of integrity. Additionally, all applications not included in the white list will experience no slowdown at all, which supports the pragmatical usability of VaultFS in contexts where both whitelisted and non-whitelisted applications can be simultaneously used.

6.2.2 JITScanner

For JITScanner, we carried out two distinct experiments in our study. Initially, we measured the performance of our solution when using conventional applications. Subsequently, we assessed the slowdown introduced by our solution for applications employing Just-in-Time (JIT) compilation, which make

(important) usage of WX pages.

For the first test, we executed multiple trials of several classic Linux command-line utilities, both with and without the presence of our LKM, and with and without the synchronous kernel-level check. Each data point reports the average execution time over 2,000 samples. The results are shown in Figure 6.8. Additionally, the slowdown caused by our module without the synchronous check is reported in Table 6.6. Our findings indicate a slowdown in the range of 7% to 13%, which we can consider acceptable.

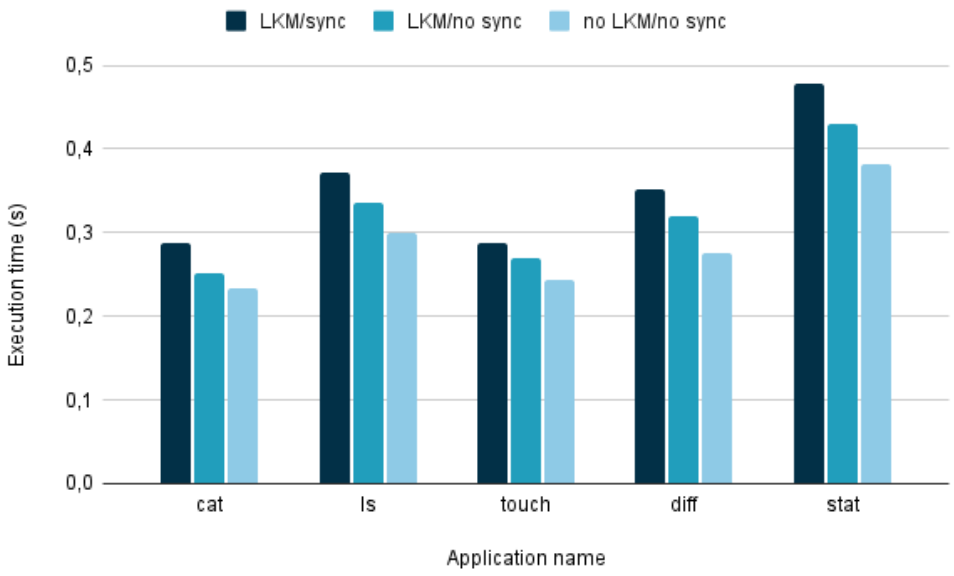

Figure 6.7: Execution time of common command-line utilities

For our second test, we selected three programming languages that are compatible with Just-in-Time (JIT) compilation: Lua, PHP, and Ruby; and used them to run a set of commonly utilized tests from the Debian benchmark game [58]. Also, to establish a performance baseline, we included the C language as a reference point. To mitigate excessive variability in our measurements, we configured the benchmark parameters to execute computationally non-trivial workloads, each with an execution time of approximately one second. Subsequently, similar to what done for the previous test, we conducted 500 iterations of the benchmarks under the following conditions: with and without the incorporation of our LKM, and with and without the synchronous level check of

the page content.

The results of these experiments are presented in Figure 7 and the performance slowdowns are reported in Table 3. Our observations are as follows:

1. in the absence of the synchronous check, no test exhibited a slowdown exceeding 5%, and in some instances, no discernible slowdown was observed at all;

2. with the synchronous check enabled, the majority of tests experienced a slowdown of less than 10%, with some exceptions noted for the PHP language-based tests.

With these results and taking into account the possibility to disable the synchronous check, which, in certain scenarios, may have a greater impact on the slowdown, our architecture introduces a negligible performance overhead on JIT-based applications.

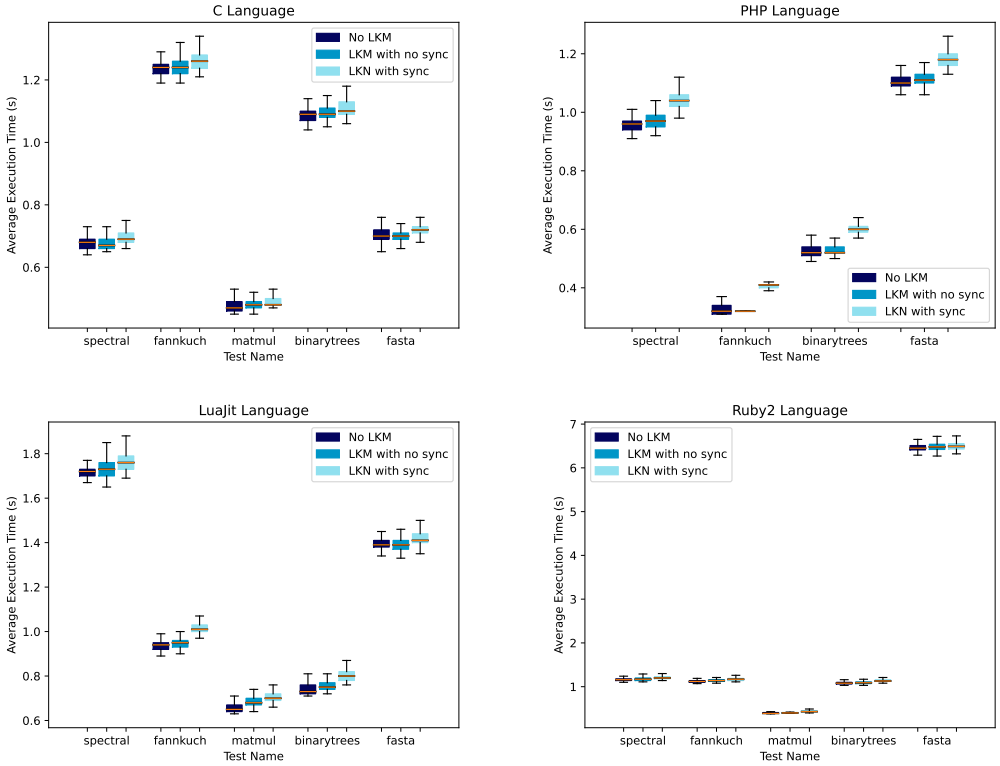

Figure 6.8: Execution time of JIT benchmarks

Table 6.6: Slowdown of common command-line applications with no synchronous page check

Application Name	Slowdown
cat	7.46%
ls	10.57%
touch	9.95%
diff	13.90%
stat	11.06%

Table 6.7: Slowdown of JIT benchmark applications

Language	Benchmark Name	No Sync Slowdown	Sync Slowdown
C	spectralnorm	0.00%	1.47%
C	fannkuchredux	0.00%	1.61%
C	matmul	2.13%	2.13%
C	binarytrees	0.00%	0.92%
C	fasta	0.00%	2.86%
LuaJit	spectralnorm	0.58%	2.33%
LuaJit	fannkuchredux	1.06%	7.45%
LuaJit	matmul	4.62%	7.69%
LuaJit	binarytrees	2.74%	9.59%
LuaJit	fasta	0.00%	1.44%
PHP	spectralnorm	1.04%	8.33%
PHP	fannkuchredux	0.00%	28.12%
PHP	binarytrees	0.00%	15.38%
PHP	fasta	0.91%	7.27%
Ruby2	spectralnorm	0.86%	3.45%
Ruby2	fannkuchredux	1.79%	4.46%
Ruby2	matmul	2.56%	10.26%
Ruby2	binarytrees	0.93%	4.63%
Ruby2	fasta	0.47%	0.62%

Additionally, it should be noted that our performance evaluation results refer to a kind of worst-case scenario for our system, as we tested it on very short-lived applications. In these cases, the cost of the initial security check on a page may not be fully offset by its subsequent accesses for instruction fetch. The slowdown is therefore expected to be less pronounced in applications where the longer overall execution time will mitigate the impact of the initial page check.

6.2.3 FlowScanner

For the performance study of FlowScanner, we focus on capturing the possible overhead introduced for managing all the tasks in the software architecture (e.g., the interception of ill-op traps and the management of master page copies). For this performance test, we executed a series of benchmarks obtained from Debian Benchmark [58] and SPEC CPU 2017 [26]. We chose SPEC CPU because it is a widely used benchmark suite for evaluating the performance impact of DBI. Additionally, we selected the Debian Benchmark as it is the only open-source benchmark we found that allows us to measure the overhead of the same application under both native execution and various JIT-based languages and strategies. In particular, for the Debian Benchmark, we present the results obtained by running the tests 500 times each, across four different languages: C, PHP, LuaJIT and Python, each employing distinct execution strategies and memory usage methods. For the test selection, we took all benchmarks which executed correctly on the native machine, with no module or engine in place. For the SPEC CPU 2017 benchmark we executed a run in its default configuration with "peak" compiler optimizations.

We compare the performance provided by FlowScanner with the ones observed when the benchmarks are run a) on the native machine with no tracing, this serves as the baseline, and when they are run b) with the tracing of the most popular DBI tools which we were able to obtain: QEMU, DynamoRIO and Pin. We selected DBI tools for comparison because they are the only publicly available solutions capable of tracing applications at the basic-block level while also supporting automatic detection of self-modifying or JIT-generated code. QEMU, DynamoRIO, and Pin were specifically chosen because they are some of the most mature and widely used DBI tools. For FlowScanner we have run a minimal synchronous (on the critical path) signature-check routine at kernel level—carried out by the Dynamic Check Engine of FlowScanner architecture—which just identifies the presence of the meterpreter shell code on the basic-block that is used by the application. In contrast, the DBI engines

Table 6.8: SPEC CPU execution time

Benchmark Name	Base Time (s)	Overhead			
		Qemu	Dynamo	Pin	FlowScanner
600.perlbench_s	422	659%	68%	117%	0%
602.gcc_s	672	435%	51%	78%	0%
605.mcf_s	906	175%	15%	20%	0%
620.omnetpp_s	614	251%	17%	30%	4%
623.xalancbmk_s	566	198%	48%	32%	0%
625.x264_s	317	920%	22%	37%	0%
631.deepsjeng_s	499	374%	23%	42%	0%
641.leela_s	567	256%	10%	26%	1%
648.exchange2_s	322	421%	8%	21%	1%
657.xz_s	3466	129%	12%	17%	0%

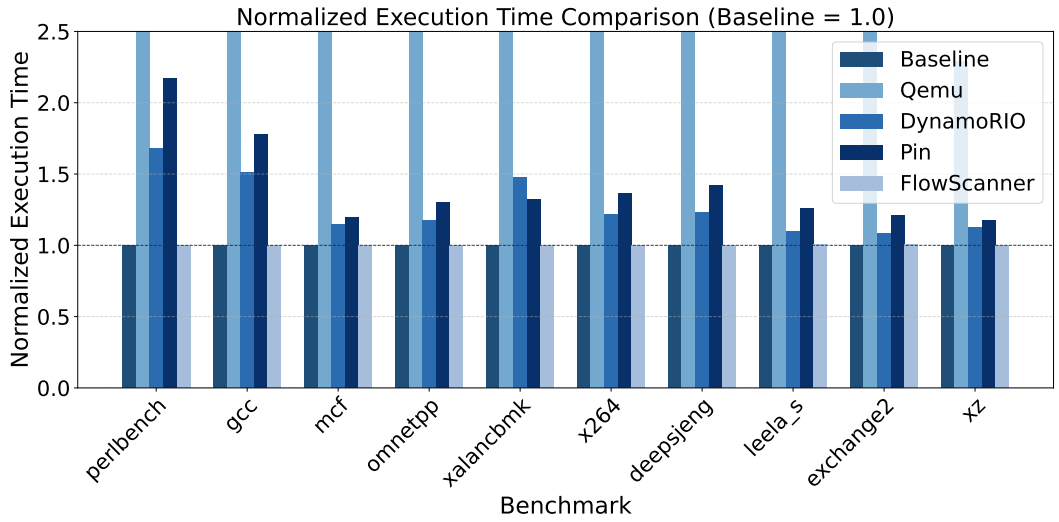

Figure 6.9: Execution Time of SPEC CPU benchmarks — QEMU exceeds the plot maximum (up to 9×)

were evaluated using their default configurations, without performing any additional analysis. This setup is unfavorable to FlowScanner, as it performs more work by analyzing each basic block, whereas the DBI engines are not running any detection logic. However, our goal was to demonstrate that the overhead introduced by DBI engines is not due to the analysis logic, but rather an inherent cost of their internal instrumentation mechanisms.

The results of the SPEC CPU run are presented in Table 6.8 and Figure 6.9. As shown, our tool exhibits no discernible slowdown under this configuration. This is likely because the benchmark tests are long-running and typically execute the same code repeatedly, minimizing overhead after the application has passed the initial few seconds of execution. In contrast, all other tools

Table 6.9: Debian JIT benchmarks execution time

Benchmark		Base Time (s)	Overhead			
Language	Test		Qemu	Dynamo	Pin	FlowScanner
LuaJit	spectral	1.65	1375%	20%	263%	0%
LuaJit	matmul	0.62	1612%	69%	785%	10%
PHP	spectral	0.29	1021%	598%	7621%	11%
PHP	binarytrees	0.55	801%	368%	4410%	0%
PHP	fasta	0.45	736%	401%	4979%	4%
PHP	fannkuch	0.32	908%	603%	7265%	8%
C	spectral	0.65	673%	59%	279%	0%
C	binarytrees	1.02	537%	83%	245%	0%
C	fasta	0.58	259%	39%	281%	3%
C	matmul	0.42	681%	33%	336%	2%
C	fannkuch	1.19	163%	12%	117%	0%
PYTHON	spectral	1.59	465%	91%	na	2%
PYTHON	binarytrees	1.00	179%	86%	na	10%
PYTHON	fasta	1.31	318%	94%	na	0%
PYTHON	matmul	1.49	377%	92%	na	0%
PYTHON	fannkuch	1.71	385%	80%	na	1%

introduce some level of performance overhead—ranging from a maximum of 9x slowdown in the worst case with QEMU to a minimum of 8% with DynamoRIO in its best case. These results highlight that, even when most basic-blocks are cached, as is typical in these types of long-running applications, DBI engines still incur overhead due to the need to invoke the interpreter between transitions across cached blocks, giving FlowScanner a definitive performance advantage.

The results of the Debian Benchmarks are shown in Table 6.9 (the Python benchmarks consistently crashed when executed with Pin, preventing us from reporting the corresponding data). As shown, FlowScanner consistently outperforms all other tools across every test and language. Its maximum overhead is 10%, whereas each of the other tools exceeds 100% in at least one case and in the worst case, Pin reaches approximately 7700% overhead on one of the PHP benchmarks. This overhead in DBI engines can be attributed to two main factors: (1) the need to perform an initial analysis of the binary before execution (and again whenever code is JIT-compiled), and (2) the initialization of their internal structures before the actual test begins. Since these benchmarks are relatively short—the longest test lasts less than 2 seconds—this setup time heavily impacts the total execution time. FlowScanner avoids this issue by

deferring analysis and initialization to the last possible moment, specifically when an executable page is materialized, and only analyzes one basic block at a time, this significantly reduces overhead. Regarding JIT-compiled code, used by PHP and LuaJIT, FlowScanner shows no additional slowdown, which aligns with expectations since its analysis process applies equally to both JIT and natively compiled code. In contrast, the other tools experience greater overhead with PHP, indicating that PHP's execution strategy is particularly challenging for these DBI engines. LuaJIT performs slightly better, with only QEMU showing notably higher overhead in comparison to the others.

6.3 Effectiveness

6.3.1 VaultFS

To evaluate the effectiveness of VaultFS, we tested our architecture against a custom-built framework designed to simulate the majority of file-writing methods commonly employed by malware and adversaries.

We developed this framework, which we named LinuxAttacker [Anonimized], due to the limited availability of ransomware samples in open databases and the challenges associated with executing these samples in our environment—i.e., most samples failed to execute correctly in our sandbox because of the miss of specific resources, like for example networking. LinuxAttacker incorporates all file-overwriting techniques identified in the available ransomware samples, as well as those described in the literature and public malware reports.

LinuxAttacker takes as input the path to the file to be overwritten, the path to the block device where the file is stored, and the root of the file system containing the file. It then attempts to overwrite the specified file using the following techniques:

- **File-based:** the file is targeted by accessing it directly using the `open()` system call and subsequently written to using the `write()`, `mmap()`, `sendto()`, and `splice()` system calls.

- **Block-based:** the file is targeted by directly overwriting the underlying block device. This operation is attempted both with and without unmounting the file system managing the device.

In our study, we executed LinuxAttacker under privilege escalation and attempted to overwrite two files: one located on a traditional Ext4 file system—marked as immutable and with access permission triplets set to 0000—and another on VaultFS. As shown in Table 6.10, all techniques succeeded on the traditional file system, whereas all were effectively blocked by VaultFS.

Technique		Ext4	VaultFS
File-based	write	✓	✗
File-based	mmap	✓	✗
File-based	sendto	✓	✗
File-based	splice	✓	✗
Block-based	no umount	✓	✗
Block-based	with umount	✓	✗

Table 6.10: Effectiveness against LinuxAttacker.

6.3.2 JITScanner and FlowScanner

As the malware dataset for this effectiveness study, we used all available ELF malwares from the malware sharing "virusshare.com" website [104], comprising approximately 60,000 samples. From this dataset, we selected all ELF executables aligned with the `x86-64` ISA supported by our disassembler. This selection left us with around 2000 samples, which we used in our analysis. To better assess the accuracy of our tool, we analyzed all samples using the online malware analysis platform "VirusTotal" [105] and recorded the antivirus (AV) responses. Based on these responses, each malware sample was assigned to its most likely family according to the majority consensus of the AV engines. This classification is summarized in Table 6.11. Malware families containing fewer than ten samples were consolidated into a "minor families" category, while an "unclassified" category was designated for malware with no family identified in the VirusTotal responses or cases where there was no consensus among the AV

Table 6.11: Families of the samples used

Family Name	Malware type	Number of samples
XMRIG-Miner	coinminer	74
Mirai	Botnet	467
Gafgyt	Trojan	1606
Dofloo	Botnet	15
chinaz	DDOS	12
Tsunami	Botnet	101
Minor Families	Various	82
Unclassified	Unclassified	329

engines. In all subsequent tests, a classification was considered correct only if the malware was assigned a signature corresponding to its identified family.

As for the signature database, we had two primary requirements: the signatures had to be code-only, as our system monitors basic-blocks, and they had to successfully match at least some malware in our dataset using some recent operating system page scanner solution offered by the literature or a Full File Scan (performing static analysis) to provide a robust baseline. Unfortunately, we could not find an existing signature database that meets these criteria or any work that defines a standard for extracting such signatures. Therefore, we decided to create the database using JITScanner and then used it to test both tools, we used JITScanner as it is the tools more suitable for signature generation due to its ability to avoid detection. We chose the signatures starting from an instruction that opens an element of the control flow graph, the we select a fixed length in bytes of the binary starting at that instruction, this is a method which is commonly used [78]. For the length, we selected 16 bytes. Also, the control flow graph elements we used refer to the content of actual executable pages that have been used at runtime—possibly after a decryption of parts of the binary. To reduce the rate of false positives, we repeated the process described above on all files present in the `/usr/bin` directory, which we refer to as goodware. We then choose our final database as all the signatures recovered from the malwares minus the signatures observed in the goodware. Finally, to measure how these signatures would perform on previously unseen samples, we divided both malware and goodware into a training set (used for signature generation) and a testing set (used for actual detection). We utilized various split percentages (25%, 50%, 75%, and 90%) to observe changes when

varying the amounts of information provided.

The True Positive Rate is reported in Table 6.12. As expected, Full File Scan has the minimal value, since it does not support the check of signatures on parts of code that are obfuscated or packed, and are dynamically rewritten in memory. At the same time FlowScanner shows comparable effectiveness to JITScanner based on signature checks at page-level granularity, when the training set percentage is 75% or higher, and is getting very close when the percentage is lower, getting around 1% less detections in the worst case.

This result shows how the advantages of checking the whole page content—as JITScanner does—tend to become less relevant in scenarios where a wider knowledge base is available—including both malware and goodware—and the distance of a sample from the ones included in the training set is likely related to more reduced parts of its structure.

At the same time, for the False Positive Rate, which is presented in Table 6.13. FlowScanner shows definitely superior capabilities compared to the page-granularity approach of JITScanner across most configurations. In particular, FlowScanner leads to a minimum of 0% false positives, while the competitor approach still suffers from a 24% false positive rate with 90% of the samples used for the training set. Notably, increasing the number of goodware samples in the training set improves the outcomes of all tools. This improvement indicates that the binaries in our dataset are generally distinct and—similarly to what we discussed before—expanding the training set with additional samples actually improves the accuracy of our signature database.

As a final test, we packed all malware samples using ShiftyLoader [88] and conducted the same detection test, applying signatures derived from the plain malware. The results are summarized in Table 6.14. Interestingly, the detection rates for JITScanner and FlowScanner increased, indicating that the packing process not only failed to obfuscate the malware effectively but also introduced signatures already present in the plain malware. This phenomenon led to improved detection rates, highlighting that the packing process reinforced the signature-based detection capabilities of our tool.

Table 6.12: True Positive Rate

Training Set Percentage	Detection type	Test Set Samples	Percentage
25%	Full File Scan	1504	12.87%
25%	JITScanner	1432	97.59%
25%	FlowScanner	1453	92.15%
50%	Full File Scan	1007	34.33%
50%	JITScanner	982	97.94%
50%	FlowScanner	970	93.71%
75%	Full File Scan	498	37.45%
75%	JITScanner	484	98.33%
75%	FlowScanner	478	96.65%
90%	Full File Scan	194	61.75%
90%	JITScanner	194	97.27%
90%	FlowScanner	183	96.17%

Table 6.13: False Positive Rate

Training Set Percentage	Detection type	TestTrue Set Samples	Percentage
25%	Full File Scan	391	0.82%
25%	JITScanner	368	35.34%
25%	FlowScanner	365	3.29%
50%	Full File Scan	261	0%
50%	JITScanner	239	25.02%
50%	FlowScanner	239	2.93%
75%	Full File Scan	131	0%
75%	JITScanner	121	23.97%
75%	FlowScanner	121	2.48%
90%	Full File Scan	52	0%
90%	JITScanner	50	24.0%
90%	FlowScanner	50	0%

Overall, the above data show how FlowScanner represents a solution providing essentially the same effectiveness as JITScanner in terms of malware identification while, at the same time, being definitely more effective in terms of reduction of the incidence of false positives. Therefore, it stands as a truly effective solution for live malware detection, at the same time, the high detection rate shows that JITScanner is effective for offline malware analysis as it was able to generate a working set of malware signatures.

Table 6.14: Packed True Positive Rate

Training Set Percentage	Detection type	Test Set Samples	Percentage
25%	JITScanner	1353	99.26%
25%	FlowScanner	1353	94.09%
50%	JITScanner	900	99.44%
50%	FlowScanner	900	95.02%
75%	JITScanner	444	99.77%
75%	FlowScanner	444	97.97%
90%	JITScanner	170	99.41%
90%	FlowScanner	170	98.23%

7. Conclusions

This thesis has explored the design and implementation of kernel-level security mechanisms aimed at improving system resilience against modern cyber threats while preserving usability and performance. Motivated by the growing sophistication of malware and the limitations of existing user-space and detection-only defenses, the work focused on solutions that operate at a higher privilege level, enabling fine-grained control over program execution and strong guarantees on data integrity.

The core contribution of this dissertation is the demonstration that kernel-mediated interception of execution and storage primitives can significantly improve both malware analysis and system resilience, without relying on specialized hardware or imposing prohibitive overhead. By intervening at critical points in the attack lifecycle: analysis, detection, and post-compromise damage prevention; the proposed solutions collectively form a layered defense strategy capable of addressing both external attackers and high-privilege insiders.

In the malware analysis phase, JITScanner introduced a novel approach to executable-page interception based on virtualized page permissions. By leveraging page-table manipulation and page-fault interception, JITScanner is able to trace the first execution of every executable page, including dynamically generated or self-modifying code, without altering the target process address space. This design makes JITScanner particularly suitable for offline malware analysis and signature generation, as it significantly reduces detectability by adversarial samples while maintaining low overhead. The shadow state machine for WX pages further extends its applicability to modern malware that relies on just-in-time decryption or runtime code generation.

Building on this foundation, the malware detection phase was addressed through FlowScanner, which increases execution tracing granularity from pages to basic blocks. By combining page-level interception with selective binary instrumentation, FlowScanner captures precise execution flow information suitable for live malware detection and signature matching. Although this higher granularity necessarily increases intrusiveness and makes the tool detectable, the design explicitly prioritizes robustness against evasion over transparency, aligning with the requirements of real-time defense systems. Experimental results show that this trade-off yields improved detection capabilities while still maintaining acceptable performance.

Recognizing that detection mechanisms can never be perfectly reliable, the final phase of the thesis focused on system resilience in compromised environments. VaultFS provides a software-only, Linux-based write-once file system that enforces strong data integrity guarantees even in the presence of attackers with root privileges. Unlike existing solutions based on detection, versioning, or specialized hardware, VaultFS adopts a secure-by-design approach that prevents unauthorized modification or deletion of protected data by construction. Its compatibility with common storage devices, support for configurable protection lifetimes, and resilience to denial-of-service attempts make it suitable for deployment in real-world critical infrastructures.

Comprehensive experimental evaluation confirms that all proposed systems achieve their security objectives with minimal performance overhead and high compatibility with existing applications and workflows. By integrating security mechanisms directly into kernel execution paths that are already performance-critical, the solutions amortize their cost and remain practical for continuous use.

Overall, this work demonstrates that operating-system–level security mechanisms can move beyond coarse-grained monitoring and probabilistic detection, providing precise execution visibility and deterministic integrity guarantees. The proposed techniques show that resilience and usability need not be conflicting goals, and that carefully designed kernel-level interventions can

offer strong protection while remaining transparent to legitimate users.

8. Acknowledgments

This thesis is the result of three years of dedication and perseverance. Along the way, I came across several challenges, but each obstacle became an opportunity to learn and improve.

I would like to express my deepest gratitude to my supervisor, Prof. Francesco Quaglia, for his invaluable guidance and support. His attention to detail and continuous commitment to excellence pushed our work to the highest possible standard. I am also sincerely thankful to Prof. Giuseppe Bianchi, who introduced me to the world of research. His contributions went beyond ideas, helping us refine how we communicate and present our work to a wider audience.

My appreciation extends to my university colleagues, who became friends along the way and made every day in the lab feel less like work and more like time spent among friends. They also played a role in cultivating a hobby that many computer scientists inevitably develop: the sudden urge to climb and ski down mountains.

I am deeply grateful to all my friends for always being there when I needed them, especially Ezio and Giovanni. We spent so many days together playing stupid games, which I will, of course, keep winning, thanks to my steep learning curve.

I would like to thank my family for their unconditional support. They never questioned my decision to leave the security of a “posto fisso” and instead encouraged me to pursue a path in research.

Finally, this milestone is not an end, but rather the beginning of a new journey.

Bibliography

[1] Oluwadamilare Harazeem Abdulganiyu, Taha Ait Tchakoucht, and Yakub Kayode Saheed. "A systematic literature review for network intrusion detection system (IDS)". In: *International journal of information security* 22.5 (2023), pp. 1125–1162.

[2] Habtamu Abie. "An overview of firewall technologies". In: *Telektronikk* 96.3 (2000), pp. 47–52.

[3] acquasecurity. `https://aquasecurity.github.io/tracee/v0.6.4/`. 2020–2023.

[4] *An overview of Microsoft Project Silica and its archive use*. `https://www.techtarget.com/searchstorage/feature/An-overview-of-Microsoft-Project-Silica-and-its-archive-use`.

[5] Theodoros Apostolopoulos et al. "Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks". In: *Future Generation Computer Systems* 116 (2021), pp. 393–405.

[6] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. "SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 7141–7158. ISBN: 978-1-939133-44-1. URL: `https://www.usenix.org/conference/usenixsecurity24/presentation/avllazagaj`.

[7] Sungha Baek et al. "SSD-insider: Internal defense of solid-state drive against ransomware with perfect data recovery". In: *2018 IEEE 38th*

International Conference on Distributed Computing Systems (ICDCS). IEEE, July 2018.

[8] Usukhbayar Baldangombo, Nyamjav Jambaljav, and Shi-Jinn Horng. "A Static Malware Detection System Using Data Mining Methods". In: *arXiv [cs.CR]* (Aug. 2013).

[9] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics". In: *Proceedings 2018 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2018.

[10] Fabrice Bellard and the QEMU Team. *QEMU: Open Source Processor Emulator*. `https://www.qemu.org/`. Accessed: 2024-03-12. 2024.

[11] Giorgio Bernardinetti, Dimitri Di Cristofaro, and Giuseppe Bianchi. "PEzoNG: Advanced Packer For Automated Evasion On Windows". In: *J. Comput. Virol. Hacking Tech.* 18.4 (2022), pp. 315–331. DOI: `10.1007/s11416-022-00417-2`. URL: `https://doi.org/10.1007/s11416-022-00417-2`.

[12] Andrew R Bernat and Barton P Miller. "Anywhere, any-time binary instrumentation". In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. New York, NY, USA: ACM, Sept. 2011.

[13] Sebastian Berrios et al. "Systematic review: Malware detection and classification in cybersecurity". In: *Applied Sciences* 15.14 (2025), p. 7747.

[14] D Bruening, T Garnett, and S Amarasinghe. "An infrastructure for adaptive dynamic optimization". In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE Comput. Soc, 2003.

[15] *Building Blocks for Data Center Cloud Architectures*. `https://www.ciscopress.com/articles/article.asp?p=2273594&seqNum=4`.

[16] Alexei Bulazel and Bülent Yener. “A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web”. In: Nov. 2017, pp. 1–21. DOI: 10.1145/3150376.3150378.

[17] Pasquale Caporaso, Giuseppe Bianchi, and Francesco Quaglia. “JITScanner: Just-in-Time Executable Page Check in the Linux Operating System”. In: *Applied Sciences* 14.5 (2024). ISSN: 2076-3417. DOI: 10.3390/app14051912. URL: https://www.mdpi.com/2076-3417/14/5/1912.

[18] Stefano Carnà et al. “Fight Hardware with Hardware: Systemwide Detection and Mitigation of Side-Channel Attacks Using Performance Counters”. In: *ACM Digital Threats Research and Practice* 4.1 (2023). ISSN: 2692-1626. DOI: 10.1145/3519601. URL: https://doi.org/10.1145/3519601.

[19] Chen Chen et al. “A systematic review of fuzzing techniques”. In: *Computers & Security* 75 (2018), pp. 118–137.

[20] Jing Chen et al. “Uncovering the Face of Android Ransomware: Characterization and Real-Time Detection”. In: *IEEE Transactions on Information Forensics and Security* 13.5 (2018), pp. 1286–1300. DOI: 10.1109/TIFS.2017.2787905.

[21] *ClamAV*. https://www.clamav.net/. 2004-2024.

[22] *Compliant WORM storage using NetApp SnapLock*. https://www.netapp.com/pdf.html?item=/media/6158-tr4526pdf.pdf.

[23] Andrea Continella et al. “ShieldFS: a self-healing, ransomware-aware filesystem”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. Ed. by Stephen Schwab, William K. Robertson, and Davide Balzarotti. ACM, 2016, pp. 336–347. URL: http://dl.acm.org/citation.cfm?id=2991110.

[24] *Control-flow integrity for the kernel*. https://lwn.net/Articles/810077/.

[25] Oracle Corporation. https://btrfs.readthedocs.io/en/latest/. 2008.

[26] CPUSpec. https://www.spec.org/cpu2017/. 2017.

[27] *CramFS*. https://docs.kernel.org/filesystems/cramfs.html.

[28] Nadia Daoudi et al. "A deep dive inside drebin: An explorative analysis beyond android malware detection scores". In: *ACM Transactions on Privacy and Security* 25.2 (2022), pp. 1–28.

[29] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. "SPIDER: stealthy binary program instrumentation and debugging via hardware virtualization". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACSAC '13. New Orleans, Louisiana, USA: Association for Computing Machinery, 2013, 289–298. ISBN: 9781450320153. DOI: 10.1145/2523649.2523675. URL: https://doi.org/10.1145/2523649.2523675.

[30] I Putu Arya Dharmaadi, Elias Athanasopoulos, and Fatih Turkmen. "Fuzzing frameworks for server-side web applications: a survey". In: *International Journal of Information Security* 24.2 (2025), p. 73.

[31] Sushant Dinesh et al. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2020, pp. 1497–1511.

[32] Abdulrahman Abu Elkhail et al. "Seamlessly Safeguarding Data Against Ransomware Attacks". In: *IEEE Trans. Dependable Secur. Comput.* 20.1 (2023), pp. 1–16. DOI: 10.1109/TDSC.2022.3214781. URL: https://doi.org/10.1109/TDSC.2022.3214781.

[33] European Union. *General Data Protection Regulation (GDPR)*. EU Regulation 2016/679 on data protection and privacy, effective May 2018. 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[34] Linux Foundation. *Perf tools support for Intel Processor Trace*. https://perfwiki.github.io/main/perf-tools-support-for-intel-processor-trace/. 2024.

[35] *Free list pointer protection*. https://lists.openwall.net/linux-kernel/2017/07/07/546.

[36] gamozolabs. https://github.com/gamozolabs/mesos. 2020.

[37] google. https://github.com/googleprojectzero/TinyInst. 2020.

[38] J.A. Gómez-Hernández, L. Álvarez González, and P. García-Teodoro. "R-Locker: Thwarting ransomware action through a honeyfile-based approach". In: *Computers & Security* 73 (2018), pp. 389–398. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2017.11.019. URL: https://www.sciencedirect.com/science/article/pii/S0167404817302560.

[39] Mark R Heckman and Roger R Schell. "Using proven reference monitor patterns for security evaluation". In: *Information* 7.2 (2016), p. 23.

[40] Sarah Heckman and Laurie Williams. "A systematic literature review of actionable alert identification techniques for automated static code analysis". In: *Information and Software Technology* 53.4 (2011), pp. 363–387.

[41] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[42] Sajad Homayoun et al. "Know Abnormal, Find Evil: Frequent Pattern Mining for Ransomware Threat Hunting and Intelligence". In: *IEEE Trans. Emerg. Top. Comput.* 8.2 (2020), pp. 341–351. DOI: 10.1109/TETC.2017.2756908. URL: https://doi.org/10.1109/TETC.2017.2756908.

[43] Kun Hu et al. "A Survey of Fuzzing Open-Source Operating Systems". In: *arXiv preprint arXiv:2502.13163* (2025).

[44] Jian Huang et al. “FlashGuard”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2017.

[45] Jian Huang et al. “FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani Thuraisingham et al. ACM, 2017, pp. 2231–2244. DOI: 10.1145/3133956.3134035. URL: https://doi.org/10.1145/3133956.3134035.

[46] IBM. *The JIT compiler*. https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=reference-jit-compiler. 1993.

[47] Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. “Combining Large Language Models with Static Analyzers for Code Review Generation”. In: *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE. 2025, pp. 174–186.

[48] Brijesh Jethva et al. “Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring”. In: *J. Comput. Secur.* 28.3 (2020), pp. 337–373. DOI: 10.3233/JCS-191346. URL: https://doi.org/10.3233/JCS-191346.

[49] Sangmoon Jung and Yoojae Won. “Ransomware detection method based on context-aware entropy analysis”. In: *Soft Comput.* 22.20 (2018), pp. 6731–6740. DOI: 10.1007/s00500-018-3257-z. URL: https://doi.org/10.1007/s00500-018-3257-z.

[50] Mohammad Sina Karvandi et al. “HyperDbg: Reinventing Hardware-Assisted Debugging”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1709–1723.

[51] Mohammad Sina Karvandi et al. “The Reversing Machine: Reconstructing Memory Assumptions”. In: *arXiv [cs.CR]* (May 2024).

[52] Edwin Kayang et al. “R-visor: An extensible dynamic binary instrumentation and analysis framework for open instruction set architectures”. In: *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM, June 2025, pp. 158–169.

[53] Amin Kharraz and Engin Kirda. “Redemption: Real-time protection against ransomware at end-hosts”. In: *Research in Attacks, Intrusions, and Defenses*. Lecture notes in computer science. Cham: Springer International Publishing, 2017, pp. 98–119.

[54] Amin Kharraz and Engin Kirda. “Redemption: Real-Time Protection Against Ransomware at End-Hosts”. In: *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*. Ed. by Marc Dacier et al. Vol. 10453. Lecture Notes in Computer Science. Springer, 2017, pp. 98–119. DOI: 10.1007/978-3-319-66332-6_5. URL: https://doi.org/10.1007/978-3-319-66332-6_5.

[55] Amin Kharraz et al. “UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware”. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 757–772. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz.

[56] Eugene Kolodenker et al. “PayBreak: Defense Against Cryptographic Ransomware”. In: Apr. 2017, pp. 599–611. DOI: 10.1145/3052973.3053035.

[57] Joxean Koret and Elias Bachaalany. *The antivirus hacker’s handbook*. John Wiley & Sons, 2015. Chap. 9.

[58] kostya. https://github.com/kostya/jit-benchmarks. 2023.

[59] Hsuan-Chi Kuo et al. “Set the configuration for the heart of the os: On the practicality of operating system kernel debloating”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.1 (2020), pp. 1–27.

[60] Nada Lachtar, Duha Ibdah, and Anys Bacha. “The Case for Native Instructions in the Detection of Mobile Ransomware”. In: *IEEE Letters of the Computer Society* 2.2 (2019), pp. 16–19. DOI: 10.1109/LOCS.2019.2918091.

[61] Nada Lachtar, Duha Ibdah, and Anys Bacha. “Toward Mobile Malware Detection Through Convolutional Neural Networks”. In: *IEEE Embedded Systems Letters* 13.3 (2021), pp. 134–137. DOI: 10.1109/LES.2020.3035875.

[62] Michael A Laurenzano et al. “PEBIL: Efficient static binary instrumentation for Linux”. In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, Mar. 2010.

[63] Arm Limited. *ARM CoreSight*. https://developer.arm.com/Architectures/CoreSightArchitecture. 2021.

[64] *Linux Extended File Attributes Tutorial*. https://www.linuxtoday.com/blog/linux-extended-file-attributes/.

[65] *Linux Kernel Heap Tampering Detection*. https://phrack.org/issues/66/15.

[66] *Linux Kernel ktime accessors*. https://docs.kernel.org/core-api/timekeeping.html.

[67] *Linux Security Module Usage*. https://www.kernel.org/doc/html/v4.14/admin-guide/LSM/index.html.

[68] Kaijun Liu et al. “A review of android malware detection approaches based on machine learning”. In: *IEEE access* 8 (2020), pp. 124579–124607.

[69] Simson Garfinkel Lorrie Faith Cranor. “Security and Usability: Designing Secure Systems that People Can Use”. In: 1st ed. O’Reilly Media, Inc., 2005, p. ix.

[70] Chi-Keung Luk et al. “Pin”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, June 2005.

[71] Boyang Ma et al. “Travelling the hypervisor and SSD: A tag-based approach against crypto ransomware with fine-grained data recovery”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Nov. 2023, pp. 341–355.

[72] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. “OmniUnpack: Fast, Generic, and Safe Unpacking of Malware”. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pp. 431–441. DOI: 10.1109/ACSAC.2007.15.

[73] Mamoru Mimura. “Evaluation of printable character-based malicious PE file-detection method”. In: *Internet Things* 19 (2022), p. 100521. DOI: 10.1016/J.IOT.2022.100521. URL: https://doi.org/10.1016/j.iot.2022.100521.

[74] Donghyun Min et al. “Amoeba: An autonomous backup and recovery SSD for ransomware attack defense”. In: *IEEE Comput. Arch. Lett.* 17.2 (July 2018), pp. 245–248.

[75] Kiran-Kumar Muniswamy-Reddy et al. “A Versatile and User-Oriented Versioning File System”. In: *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. FAST ’04. San Francisco, CA: USENIX Association, 2004, 115–128.

[76] Stefan Nagy and Matthew Hicks. “Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 787–802.

[77] Stefan Nagy et al. “Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1683–1700.

[78] Neo23x0. `https://github.com/Neo23x0/yarGen`. 2020.

[79] Subash Neupane et al. “Explainable intrusion detection systems (x-ids): A survey of current methods, challenges, and opportunities”. In: *IEEE Access* 10 (2022), pp. 112392–112415.

[80] Nomade040. `https://github.com/Nomade040/length-disassembler`. 2020.

[81] Jisung Park et al. “RansomBlocker”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. New York, NY, USA: ACM, June 2019.

[82] Alessandro Pellegrini. `https://github.com/alessandropellegrini/lend`. 2016.

[83] Mario Polino et al. “Measuring and defeating anti-instrumentation-equipped malware”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Lecture notes in computer science. Cham: Springer International Publishing, 2017, pp. 73–96.

[84] *Preboot Execution Environment (PXE) Specification v2.1*. `http://www.pix.net/software/pxeboot/archive/pxespec.pdf`.

[85] *Randomized slab caches*. `https://lwn.net/Articles/938246/`.

[86] Vaibhav Rastogi et al. “Cimplifier: automatically debloating containers”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 476–486.

[87] Chetan Ravi et al. “Beyond the Firewall: Implementing Zero Trust with Network Microsegmentation”. In: *Nanotechnology Perceptions* 21 (2025), pp. 560–578.

[88] Michele Salvatori et al. “ShiftyLoader: Syscall-free Reflective Code Injection in the Linux Operating System”. In: *Proceedings of the 8th Italian Conference on Cyber Security (ITASEC 2024), Salerno, Italy, April 8-12, 2024*. Ed. by Gianni D’Angelo, Flaminia L Luccio, and Francesco Palmieri. Vol. 3731. CEUR Workshop Proceedings. CEUR-WS.org, 2024.

[89] Nolen Scaife et al. “CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data”. In: *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 303–312. DOI: 10.1109/ICDCS.2016.46. URL: https://doi.org/10.1109/ICDCS.2016.46.

[90] *seL4*. https://sel4.systems/.

[91] *SElinux*. https://docs.kernel.org/admin-guide/LSM/SELinux.html.

[92] *SElinux Policies*. https://github.com/SELinuxProject/refpolicy.

[93] Saiyed Kashif Shaukat and Vinay J. Ribeiro. “RansomWall: A layered defense system against cryptographic ransomware attacks using machine learning”. In: *2018 10th International Conference on Communication Systems Networks (COMSNETS)*. 2018, pp. 356–363. DOI: 10.1109/COMSNETS.2018.8328219.

[94] Radu Sion and Yao Chen. “Fighting Mallory the insider: Strong write-once read-many storage assurances”. In: *IEEE Trans. Inf. Forensics Secur.* 7.2 (Apr. 2012), pp. 755–764.

[95] *SquashFS*. https://docs.kernel.org/filesystems/squashfs.html.

[96] Darko Stefanović et al. “Static code analysis tools: A systematic literature review”. In: *Ann. DAAAM Proc. Int. DAAAM Symp.* Vol. 31. 1. 2020, pp. 565–573.

[97] Jakapan Suaboot et al. “Sub-curve HMM: A malware detection approach based on partial analysis of API call sequences”. In: *Comput. Secur.* 92 (2020), p. 101773. DOI: 10.1016/J.COSE.2020.101773. URL: https://doi.org/10.1016/j.cose.2020.101773.

[98] Kul Prasad Subedi et al. “RDS3: Ransomware defense strategy by using stealthily spare space”. In: *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017, Honolulu, HI, USA, November 27 - Dec. 1, 2017*. IEEE, 2017, pp. 1–8. DOI: 10.1109/SSCI.2017.8280842. URL: https://doi.org/10.1109/SSCI.2017.8280842.

[99] Sysdig. *The Falco project*. https://falco.org/. 2016–2023.

[100] Yuki Takeuchi, Kazuya Sakai, and Satoshi Fukumoto. “Detecting Ransomware using Support Vector Machines”. In: *The 47th International Conference on Parallel Processing, ICPP 2018, Workshop Proceedings, Eugene, OR, USA, August 13-16, 2018*. ACM, 2018, 1:1–1:6. DOI: 10.1145/3229710.3229726. URL: https://doi.org/10.1145/3229710.3229726.

[101] *The slab and protected-memory allocators*. https://lwn.net/Articles/753154/.

[102] Amit Vasudevan and Ramesh Yerraballi. “SPiKE: engineering malware analysis tools using unobtrusive binary-instrumentation”. In: *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*. ACSC ’06. Hobart, Australia: Australian Computer Society, Inc., 2006, 311–320. ISBN: 1920682309.

[103] R. Vinayakumar et al. “Evaluating shallow and deep networks for ransomware detection and classification”. In: *2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2017, Udupi (Near Mangalore), India, September 13-16, 2017*. IEEE, 2017, pp. 259–265. DOI: 10.1109/ICACCI.2017.8125850. URL: https://doi.org/10.1109/ICACCI.2017.8125850.

[104] virus_share. https://virusshare.com/torrents. 2020.

[105] VirusTotal. https://www.virustotal.com. 2024.

[106] VirusTotal. *Yara rules*. https://virustotal.github.io/yara/. 2014–2023.

[107] Peiying Wang et al. "MimosaFTL". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. New York, NY, USA: ACM, Mar. 2019.

[108] Ruoyu Wang et al. "Ramblr: Making reassembly great again". In: *Proceedings 2017 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2017.

[109] Carsten Willems, Felix C. Freiling, and Thorsten Holz. "Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida, USA: Association for Computing Machinery, 2012, 179–188. ISBN: 9781450313124. DOI: 10.1145/2420950.2420979. URL: https://doi.org/10.1145/2420950.2420979.

[110] *Write-Once-Read-Many (WORM) tamper proof technology*. https://www.nexusindustrialmemory.com/write-once-read-many/.

[111] Yanfang Ye et al. "Intelligent file scoring system for malware detection from the gray list". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 1385–1394.

[112] Cagatay Yucel and Ahmet Koltuksuz. "Imaging and evaluating the memory access for malware". In: *Digit. Investig.* 32 (2020), p. 200903. DOI: 10.1016/J.FSIDI.2019.200903. URL: https://doi.org/10.1016/j.fsidi.2019.200903.

[113] Xiaogang Zhu et al. "Fuzzing: a survey for roadmap". In: *ACM Computing Surveys (CSUR)* 54.11s (2022), pp. 1–36.