VFSMon: an Innovative Reference Monitor in Linux

Edoardo Manenti[1,2], Pasquale Caporaso[1,2], Giuseppe Bianchi[1,2] and Francesco Quaglia[1]

[1] *UNIVERSITÀ DEGLI STUDI DI ROMA "TOR VERGATA" | Via Cracovia, n.50 - 00133, Roma (RM)*
[2] *CNIT NAM LAB | Via del Politecnico, n.1 - 00133, Roma (RM)*

Abstract

This paper introduces VFSMon, a path-based reference monitor that selectively protects files and device files in Linux. VFSMon operates at the Virtual File System (VFS) layer, ensuring independence from underlying file systems. VFSMon's key features include selective write access restrictions, time-based protection with resilient time management, and an embedded credential mechanism for configurable control over privileged users like root. Furthermore, VFSMon offers place-holding paths, that can be exploited for common data management workflows based on production of (backup) files and their unmodificability for specific time windows. By protecting both pathname and device file access, VFSMon enhances security while supporting flexibility, providing a practical solution for safeguarding critical data against unauthorized modifications.

Keywords

Linux, Virtual File System, Kernel Module, Kernel Probes, Reference Monitor, Data Protection,

1. Introduction

A wide range of critical applications rely on the permanent recording of data. This need is especially evident in contexts where backup images of virtual machines, file systems, and databases are fundamental for responding to adverse events and ensuring that applications and services can be correctly restarted.

At the same time, in most common scenarios, simply relying on offline data copies is insufficient to provide urgent data accessibility when specific events occur. As a result, data are often accessible online, albeit behind protection mechanisms such as VPNs [1] and authentication protocols [2].

However, this approach increases the level of risk since the data are more exposed. Once an attacker—such as a malicious insider—can manipulate applications on the operating system instance where the data reside, the complete removal or encryption of that data becomes a genuine possibility.

This problem has been long studied in the literature, and several solutions have been proposed to attempt the prevention of unwanted data modifications. A few of them are suited for specific attacks, like for example Ransomware and its typical data update pattern [3]. Other proposals operate at the level of the file system where data are stored, and exploit additional attributes for files and directories, like for example the "immutability" flag (+i) in ext2/3/4 filesystems.

Additional, proposals rely on Write-Once Read-Many (WORM) technology for the devices where data are really stored [4, 5, 6]. Also, other solutions exploit the concept of reference monitor [7], and embed within the reference monitor policies that prevent any thread, running on behalf of any user, to overwrite or delete data in the file system.

While each of these solutions has its own advantages, they also show limits. In particular, WORM technology involves non-reusable storage blocks, protection based on data update patterns is essentially a statistical technique, additional flags in the file system can be anyhow reset by threads running with effective root-ID, and common reference monitors are not enough flexible to enable specific properties in the data protection system. In this article we present an alternative solution named VFS-Monitor

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
edoardo.manenti@uniroma2.it (E. Manenti); pasquale.caporaso@uniroma2.it (P. Caporaso); giuseppe.bianchi@uniroma2.it (G. Bianchi); francesco.quaglia@uniroma2.it (F. Quaglia)
http://netgroup.uniroma2.it/GiuseppeBianchi/ (G. Bianchi); https://francescoquaglia.github.io/ (F. Quaglia)
0009-0008-3503-1209 (E. Manenti); 0009-0001-0552-7894 (P. Caporaso); 0000-0001-7277-7423 (G. Bianchi); 0000-0002-5616-7980 (F. Quaglia)

(VFSMon), which combines features that characterize the different aforementioned techniques, while at the same time counteracting their disadvantages.

VFSMon is a software architecture we developed for integration in the Linux kernel, and is fully based on the Linux Kernel Module (LKM) technology. It can be configured like a traditional reference monitor, requiring that update operations on files and directories can be prevented at all (depending on the security policies defined by the security administrator). At the same time it also offers additional mechanisms, which permit the setup of common workflows in data management, like the possibility to include in the file system new data images (like database backups) which can be protected from updates along a predefined time window—this is the classical workflow exploited for enabling the deletion of older data copies that become no longer relevant at some point in time. All these activities can be done through VFSMon with no need for any reconfiguration of parameters at run time, and with the impossibility to make changes in the managed policies, even by root-ID threads.

At the same time, VFSMon can also be setup with a less critical operating mode, enabling the possibility of easy reconfigurations of the data protection rules, still relying on proper credentials that do not coincide with the ones of the root user of the operating system—it therefore supports the notion of security administrator, in a similar manner to the one used by commercial products like [8]. Also, VFSMon is usable with any kind of file system to be mounted, being it an object essentially working at the virtual level of the overall file system architecture, which expands its usability significantly.

The remainder of this article is structured as follows. In Section 2 we discuss related work. In Section 3 we present VFSMon. A study on the overhead introduced by the proposed solution is provided in Section 4. Conclusions are discussed in Section 5.

2. Related Work

One of the main problems when dealing with data kept at the file system level are Ransomware attacks [9, 10]. Solutions oriented to tackling this type of attacks are either based on the inclusion of file recoverability at the file system level [11, 12] or on the delayed flushing of updated data from the page-cache to the actual hard drive [3]. The recovery support, or the suspended flush, are maintained until threads are still under judgment of their activities, and in any case not beyond a time limit. Our VFSMon proposal works in a different manner, since it can support the unmodificability of data for time windows whose length is independent from thread activities in the system. This allows supporting specific data management workflows in a more suitable manner.

Another technique for preventing data alteration or deletion, has traditionally relied on WORM (Write-Once Read-Many) device technologies [4, 5, 6]. Our VFSMon solution stands apart from these approaches, since it purely operates at the software level, particularly at the level of the virtual file system, eliminating the need for any specialized hardware investment.

A recent solution able to tackle both Ransomware and insider attacks is VaultFS [13], which is a Linux oriented file system that enables a single session write activity on files. Each file can therefore remain stable after it is created, e.g., by some backup generation system. One core difference between VaultFS and our VFSMon solution is that the former operates on a file system that needs to reside on a physical device—and that is managed by a specific driver, while VFSMon works on any file/directory belonging to whichever file system, including the ones that are embedded within device files residing on a different file system (like those manageable via the `-o loop` driver). Furthermore, working at the level of the virtual file system, our solution stands above any file system specific driver, being therefore compatible with differentiated data management solutions offered at the level of the operating system.

The Linux kernel also offers FS-VERITY [14], a feature that provides transparent integrity and authenticity protection for read-only files. It operates by constructing a Merkle tree over the content of a file, allowing the system to verify any read operation against this tree to detect unauthorized modifications that might have occurred. This mechanism ensures that if any part of the file has been altered, the verification process will fail, thereby preventing the compromised data from being used. While this solution is extremely useful for blocking the usage of programs—like libraries—that can have

Solution	Protects from intruder (root-level) adversary	Extra HW needed	Time-based protection	File system independent	Dynamic reconfiguration
Ransomware-specific [11, 12, 3]	No	No	No	No	Limited
WORM-based [4, 5, 6]	Yes	Yes	No	No	No
FS-VERITY [14]	Detection Only	No	No	No	Complex
SELinux [7]	Yes	No	No	Yes	Complex
VaultFS [13]	Yes	No	Yes	No	Yes
VFSMon (this article)	Yes	No	Yes	Yes	Yes

Table 1
Comparison of different data protection solutions

been altered (modified) by whatever attack, it doesn't block modifications, which can anyhow take place. VFSMon offers protection against modification, for time windows that can be predefined by the security administrator (ideally they can have infinite length).

Security-Enhanced Linux (SELinux) [7] is a robust security framework that provides a mechanism for supporting access control security policies, including mandatory access control (MAC). It operates on a whitelisting approach, enforcing security through comprehensive policies that define what actions are explicitly permitted. In SELinux, everything not expressly allowed is denied by default. On the opposite side, VFSMon relies on a blacklisting approach, thus enabling the easy definition of borders for security. Also, SELinux's policies are generally static—this is the way it uses in order to guarantee MAC—which can be a drawback in environments where security requirements can change—like for example depending on the age of a specific file. VFSMon, on the other hand, also offers time-based protection for both files and directories. Hence, administrators can easily enforce access control during specific time frames, making VFSMon highly adaptable to varying operational contexts. Additionally, a feature that distinguishes VFSMon is that is can be setup to manage protection for file system paths of objects that do not already exist. Once they will be created, they will keep their update protection for the predetermined time window.

Finally, being VFSMon oriented to MAC protection, it can prevent threads that run with root-ID to carry out any operation that can damage the content of critical (hence protected) data. Several supports for Access Control List (ACL) offered by common implementations of file systems—like it occurs for example for `ext4`—do not reach this level of operations, since their setup can be anyhow subverted by root-ID threads running in the system.

Table 1 summarizes a high-level comparison of the discussed solutions.

3. VFSMon

3.1. Architectural Overview

As mentioned, the architecture of VFSMon is fully embedded within a LKM. In order to gain control when specific calls to the services of the virtual file system are issued, VFSMon exploits the KPROBE support offered by Linux. In particular, upon inserting the LKM, it installs probes on specific points in the kernel in order to intercept the occurrence of specific events.

This part of the architecture directly exploits results that are offered by Linux to explicitly support security frameworks. In particular, the probes installed by VFSMon are placed on all the hooks already present in Linux for supporting the Linux Security Module (LSM) technology, which are destined to file system management. These hooks can be seen in Table 2, where we also show the ret-probes we installed, and synthesize their activities. Although we used ret-probes, our design and implementation exploits both the entry and exit interceptions of the services offered by ret-probes.

The objective of the interception that is carried out by our kernel probes, applied to the listed hooks, is the one of determining if an operation that is requested at the file system level is compatible with the current file-system management state that is kept by VFSMon.

Hooked Function ↓ **Kretprobe**	**Description of probe activities**
`security_file_open` ↓ `krp_security_file_open`	Intercepts file opens on protected paths. If a file is opened for writing and is protected, the operation is blocked and logged.
`security_inode_rename` ↓ `krp_security_inode_rename`	Intercepts rename operations. Blocks and logs attempts involving protected paths, ensuring no bypass through renaming.
`security_inode_unlink` ↓ `krp_security_inode_unlink`	Intercepts file deletions. Blocks and logs removal attempts on protected files.
`security_inode_rmdir` ↓ `krp_security_inode_rmdir`	Intercepts directory removals. Blocks and logs attempts to remove protected directories.
`security_inode_create` ↓ `krp_security_inode_create`	Intercepts file creation in protected directories. Blocks and logs unauthorized creations.
`security_inode_mkdir` ↓ `krp_security_inode_mkdir`	Intercepts directory creation in protected directories. Blocks and logs these attempts.
`security_inode_link` ↓ `krp_security_inode_link`	Intercepts hard link creation. Blocks and logs operations involving protected files or directories.
`security_inode_symlink` ↓ `krp_security_inode_symlink`	Intercepts symlink creation referencing protected paths. Blocks and logs these attempts to prevent indirect access.

Table 2
A summary of the hooked LSM functions and their corresponding kretprobe handlers.

Such a state is characterized by a set of path-names. These path-names are not embedded in any real file-system structure, hence they do not require any management at the level of i-nodes and i-node caches. Rather, they are maintained by VFSMon in an apposite memory area. Currently, for the organization of this area, we have exploited hash-tables, which are already supported by the Linux kernel, in order to provide non-time-consuming operations when the search of a specific path-name needs to be carried out by the kernel probes of VFSMon.

These path-names are included in a set—which we simply refer to as Protected-Paths—and the passage in any of the above listed hooks—hence in any of the kernel probes—can take place with no real intervention by VFSMon only if the file/directory name involved in the requested operation—the one that leads to activating the hook—is not actually registered in the Protected-Paths set kept by VFSMon.

If the path-name is registered, then any attempt to execute file-system operations that can provide the ability to update (e.g. write) on the file or the directory is denied by VFSMon. For example, the opening of an I/O session on a file with write capability is denied. This is the baseline mechanism offered by VFSMon in terms of data protection, but as we will discuss in the successive sections, additional facilities are supported to enable the effective use of VFSMon when specific workflows, in terms of data management, need to be supported.

The Protected-Path set can be configured in VFSMon in two different manners:

- Purely at startup: in this case VFSMon is configured as a pure reference monitor offering pure MAC support for the access to files/directories, and no user is permitted to execute any action in terms of modifications (insertions/deletions) of the Protected-Paths set. This set is therefore hard-coded in the setup data of VFSMon;
- At runtime: the reconfiguration can be put in place—in particular VFSMon supports it via an `ioctl(...)` service.

However, while the scenario offering runtime reconfiguration does not represent a pure MAC operating mode, in VFSMon the reconfiguration can be put in place only relying on a couple of keys $< K_1, K_2 >$—of which K_2 needs to be passed in input to the `ioctl(...)`—such that:

- K_1 is the ID of a specific user—not necessarily coinciding with the root-user—which was setup in the LKM at its startup. This ID must correspond with the actual user-ID of the thread that runs the `ioctl(...)` command, otherwise any reconfiguration is denied by VFSMon by default.
- K_2 is a password, whose encrypted version is maintained internally by VFSMon.

By the above description, even under the scenario where the reconfiguration based on the `ioctl(...)` is supported (i.e., when VFSMon is not under pure MAC operations), any possible attacker that acquired the ID of the enabled user needs to have already tampered the usage of the `setuid(...)` service, hence requiring to tamper some other application running under (effective)root-ID. This anyhow provides an additional level of protection, which is combined with the protection offered by the encrypted password K_2 kept by VFSMon. Overall, the $< K_1, K_2 >$ couple identifies that operations are executed by a specific security administration user, which can operate reconfigurations only if non-pure MAC operating mode has been setup for VFSMon.

As hinted before, a file/directory used at the virtual file system level is managed via specific security policies offered by VFSMon only if it is included in Protected-Paths. However, this set of protected file/directory names has no direct relation with the actual data representing the file, and its i-node information. In particular, any information kept by VFSMon is fully external to the actual file systems where protected stuff is located.

Concerning this point, one core aspect that characterizes modern virtual file system architectures is that a same file content can be reachable by exploiting several different path-names. This is for example what takes place when using hard-links. In our kernel probes, we prevent the possibility to install a new hard-link to an existing file whose path-name is currently kept in Protected-Paths. This only leaves out of protection files which already have some unprotected hard-link in the file system when VFSMon starts protecting another hard-link to the same file. Also, the protection offered by VFSMon for avoiding the installation of hard-links also copes with privilege escalation, since such installation is prevented for (effective)root-ID threads too.

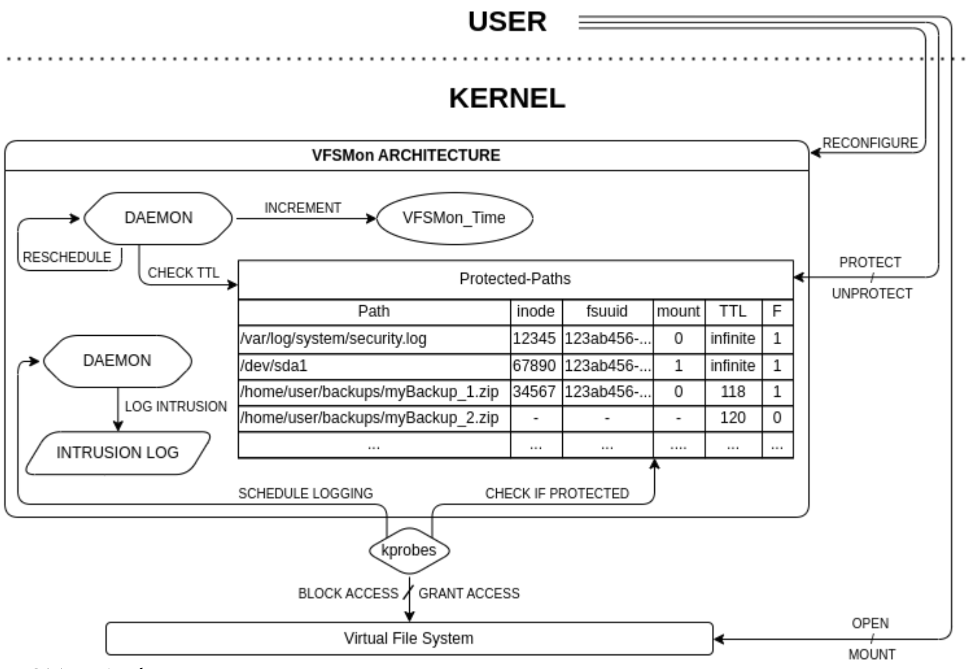
However, another aspect appears to be more critical in relation to the possibility to rely on multiple path-names for reaching a same content. It is related to bind-mount operations that can be executed by the root-ID user. These are not based on hard-links, but rather they rely on the classical `mount(...)` system call. To cope with this aspect, we installed a kernel probe also on the `mount(...)` system call service offered at kernel level. Each time it is invoked, a check is performed by VFSMon to determine if the new path for reaching stuff in the bind-mounted file system can lead to reach an object that already has its corresponding path in the Protected-Paths set. If this is the case, the new path to be protected is automatically inserted by VFSMon in Protected-Paths, and is also automatically removed when the unmount of the original bind-mount operation takes place. We think this kind of protection is relevant still in relation to privilege escalation based attacks, when some (effective)root-ID thread can operate calls to the `mount(...)` service offered by the kernel.

Additionally, considering the possibility of working with namespaces at the level of the virtual file system, the Protected-Paths offered by VFSMon is used, in terms of application of protection policies to the corresponding files/directories, independently of the current namespace the applications are working in—this is how we have designed our kernel probes. This automatically leads to the possibility to exploit a single setup of the content of the Protected-Paths set to actually protect the same files/directories that stand in different containers managed by the underlying operating system kernel.

In Figure 1 we show the architecture of VFSMon, details of which will be further discussed in the following sections.

3.2. Time Passage and Protected-Path Placeholding

In many general purpose workflows, some of the sensible data that was protected at a time t may be not sensible at a time $t + k$ and would just result in a waste of memory given the fact that removal would not be allowed. To cope with this scenario, a cardinal aspect of our system is indeed in the concept of "time to live" (TTL) associated with path protection. Essentially, it represents the time span in which



Protected-Paths

Path	inode	fsuuid	mount	TTL	F
/var/log/system/security.log	12345	123ab456-...	0	infinite	1
/dev/sda1	67890	123ab456-...	1	infinite	1
/home/user/backups/myBackup_1.zip	34567	123ab456-...	0	118	1
/home/user/backups/myBackup_2.zip	-	-	-	120	0
...	...	...	....	...	...

Figure 1: VFSMon Architecture

the path needs to stay protected and the file/directory associated with it cannot be modified. This TTL information is added by VFSMon to the protection metadata associated with a path in the used data structures. Of course it is still possible to set the time as infinite, which means that, until VFSMon is up, the file/directory will never be modifiable or deleted unless it is unprotected explicitly by the VFSMon security administrator.

However, in relation to the actual means for measuring the passage of time in VFSMon, our objective is the one of defending ourselves from attackers that are assumed to have escalated privileges—hence they can run (effective)root-ID thread. Therefore they can even be able to change date and time of the system.

Considering this aspect, we decided not to rely on any retrieval of time information to avoid creating security weaknesses and introduce possible protection bypasses. Hence, we decided to include in VFSMon a kernel daemon setup by the same VFSMon initialization block. The daemon basically triggers itself at regular intervals, exploiting relative-timeouts along the actual time passage via the High-Resolution-Timer (HRT) interrupt architecture. The only tasks run by this daemon are 1) the update of a VFSMon-internal counter that would then represent the VFSMon time and 2) the loop on Protected-Paths checking for TTL and eliminating paths when their TTL expires.

However, VFSMon not only offers the possibility to rely on TTL for reporting a file/directory into the traditional state in terms of its management—with no update protection still enforced. Rather, it also supports the management of a protected path associated with a file not yet existing. In particular, a protected path can be marked in its metadata as a *place-holding* one. Each of these paths is anyhow associated with a TTL (infinite or not), and will be protected for avoiding the opening of I/O sessions with write capabilities. However, this protection will start at the time instant where the first write session on the file is opened (and is intercepted by our probes). This means that a single session for updating the file is guaranteed to be supported, and after setting it up, new data on the file can only be placed passing through that specific I/O session. Combining this capability of VFSMon with the management of TTL enables the user to put in place workflows where specific files are allowed to be created (and populated) for keeping critical data that are guaranteed to be unmodifiable for a selected time window. This is the classical workflow related to backup procedures (of databases, virtual clusters, etc.) on an common operating site.

3.3. Managing Devices

In a modern data management system, the presence of block devices, including logical ones—like, regular files representing the device file of, e.g., virtual machine or container file systems—is crucial. Hence, providing support for the management of block devices and device files, according to their common usage—they offer file systems to be mounted—is another interesting objective.

By construction, in VFSMon a device (or device file) that cannot be written since its path-name is kept in Protected-Path, cannot be opened in write mode by any driver. However, this scenario is somehow restrictive of the real usage one can imagine for the hosted file system. In particular, a more flexible scenario would be the one where the device (or device file) is protected from updates that can occur on the file system where it is recorded, but operations in its hosted file system can anyhow occur.

VFSMon offers the possibility to support such flexible scenario. In particular, an element in Protected-Path is also marked with an additional flag that tells if a mount operation targeting that device file can be executed, also considering write-mode while mounting. This flag is managed by our probes of the mount service (see Figure 1). Hence, when the actual open operation takes place in the device driver, we recognize that the execution flow has passed through the mount task and we can enable file system mounting.

Clearly, the content of Protected-Paths still applies to files in the file system hosted on the device (or device file). Hence, VFSMon supports the scenario where file systems hosted on devices (or device files) are mounted because of the activation of, e.g. containers, and their internal content still undergoes the protection policies natively offered by VFSMon.

At the technical level, this means that these devices are only accessible through the operating system page-cache, but are not accessible as common streams on the file system where they reside. This is an additional feature that distinguishes VFSMon from common file management services offered by operating systems.

3.4. Security Assessment

Based on the explained details, we can say that our system, once in operation, succeeds in preventing any changes to protected files/directories by ensuring integrity for the required time (see TTL in Section 3.2). Integrity is guaranteed also in face of arbitrary commands or software execution by any (effective)root-ID attacker, since VFSMon supports MAC policies.

This is possible also because of some measures that have been taken for a release of VFSMon, such as not providing a `cleanup_module` function for a possible attacker to unmount the module under privilege escalation.

At the same time, the operations by VFSMon are guaranteed to work correctly unless the kernel probes it adopts are removed, or its internal modules fall under installation of additional probes that operate on them—changing their behavior. This requires anyhow tasks that cannot be executed, even under privilege escalation, except if a malicious Linux kernel module making these tasks is mounted. To protect against this scenario, VFSMon can be used under secure-boot constraints, leading to the impossibility to mount arbitrary Linux kernel modules that are not guaranteed to be trusted.

4. Run-time Cost

In this section, we present experimental data illustrating the run-time overhead introduced by VFSMon. We first describe the testing methodology, then we discuss the obtained results.

4.1. Testing Methodology

We constructed a test framework using Python and C, capable of timing file operations using the Time Stamp Counter (TSC) for cycle-accurate measurements. We measured the time taken for file `open(...)` operations in various modes (`O_RDONLY`, `O_WRONLY`, `O_CREAT | O_TRUNC | O_WRONLY`) both with and without VFSMon while running differentiated workloads by varying the number of threads and protected paths.

At first we conducted a baseline test without the presence of VFSMon, measuring the inherent latency of file operations under normal conditions and then, after mounting the module, we run the tests again, saving the new measurements. In particular, each test configuration was repeated 200 times, with each repetition performing 1000 iterations of the aforementioned file operations. These iterations were conducted while varying the size N of the set of elements kept in Protected-Paths. In particular, we used 0, 10, 100, 200, 300, 400 and 500. Furthermore, we also varied the thread count (1, 2, 4, 8) to assess the impact of concurrent file system operations intercepted and managed by VFSMon. In relation to this aspect, we recall that concurrency is managed in VFSMon with the usage of read/write spinlocks, in order to manage access to its internal structures. Hence, no blocking service is used.

In all the tests each thread targets always the same file for performing the `open(...)` operation. This choice has been motivated in order to gather latency samples related to a scenario where the device blocks that keep data/metadata required for opening the file (e.g. for performing the lookup at the level of the file system) are highly likely kept in the page-cache offered by the Linux kernel. This allows assessing the run-time cost by VFSMon in the scenario where actual I/O interactions with the device hosting the file system are avoided (thanks to the page-cache), which allows a better characterization of the potential intrusiveness of VFSMon operations.

4.2. Results

Now we proceed to present the results for the aforementioned tests by illustrating the distribution of operation times in terms of clock cycles across different thread counts and numbers of protected files. All the tests we are going to describe have been carried out on a machine equipped with 12th Gen Intel i7-12700H (10 VCPUs allocated to the used VM out of the 20 total of the host machine) @ 4.6GHz, and 8Gb (allocated out of the 16Gb of the host machine).

In Figure 2 we report the latency of open operations in read-mode. Since in the access control carried out by VFSMon checks are only triggered on write-mode opens, read operations experience negligible overhead, even with an increasing number of protected files. The data indicates consistent median read times across all configurations, with only minor differences observed between the absence or presence of our kernel module. We can then see the illustration of the latency of write-mode operations in Figure 3 where, although access control checks are triggered, the overhead remains low due to the fact that VFSMon employs a hashmap-based path-matching mechanism during its security checks, which scales efficiently with the number of protected files. Lastly, in Figure 4 we present the results for file creation operations. The create mode involves opening files with flags `O_CREAT | O_TRUNC | O_WRONLY`, triggering both write-mode access control checks and additional file system operations. This use case shows how the overhead by VFSMon is further reduced, given the larger amount of intrinsic operations that are executed at the level of the actual file system, which leads to reduce the percentage incidence of VFSMon operations.

5. Conclusions and Future Work

In this paper, we introduced VFSMon, a path-based reference monitor that operates at the Virtual File System layer to deliver a flexible and dynamically configurable security solution for Linux. By combining selective write-access restrictions, time-based protection windows, and an embedded credential mechanism, VFSMon addresses several critical shortcomings of traditional reference monitors, offering a more practical and fine-grained approach to safeguarding both file and device file access. In addition, the novel concept of place-holding paths facilitates common data management workflows, such as controlled backup generation and enforced file immutability over defined periods. Our evaluation demonstrates that VFSMon achieves these enhancements at a reasonable performance toll with respect to the Linux baseline. This makes it well-suited for environments where security requirements are high

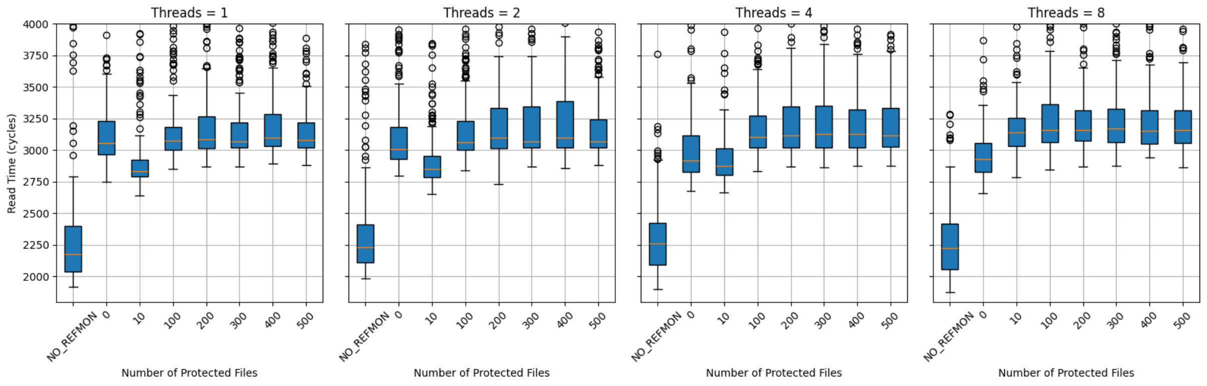


Figure 2: Read-mode open time cycles by number of protected files for each thread count

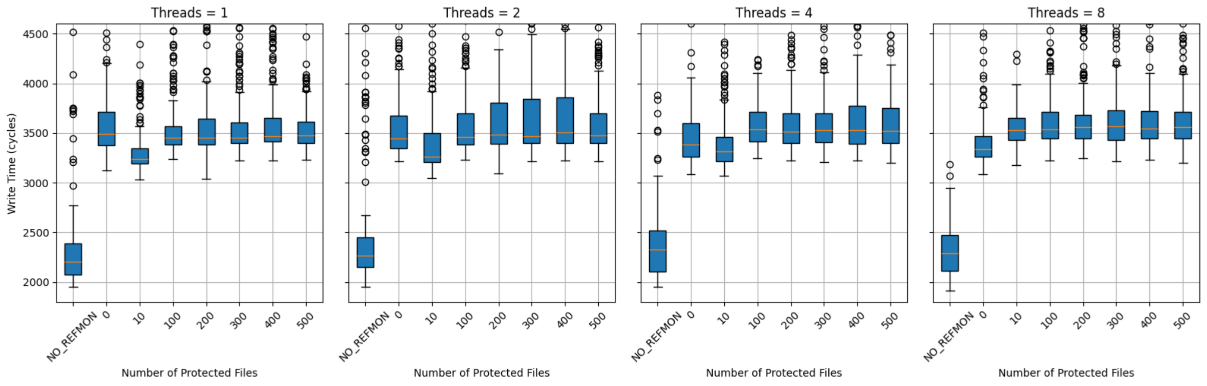


Figure 3: Write-mode open time cycles by number of protected files for each thread count

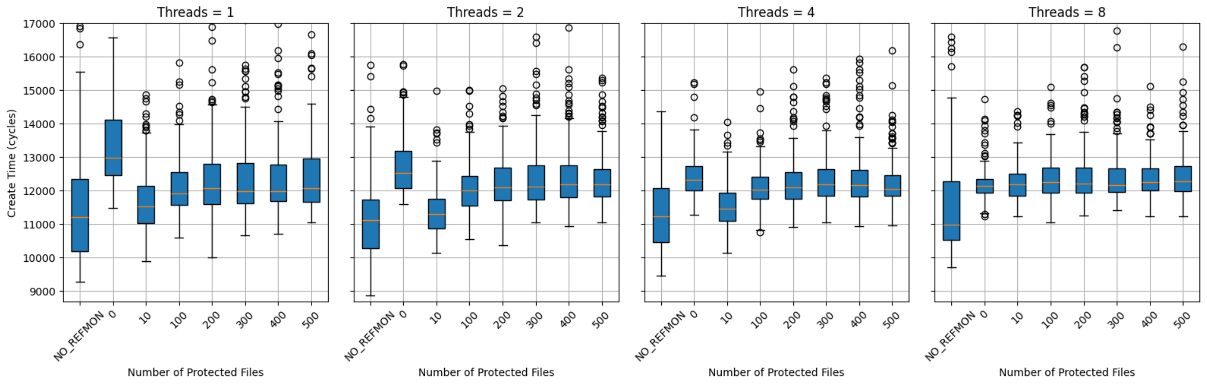


Figure 4: Create time cycles by number of protected files for each thread count

but must remain adaptive to changing operational conditions. Future work includes extending policy configuration options and integrating user-friendly management tools.

Acknowledgments

This work was supported by Agenzia per la cybersicurezza nazionale under the programme for promotion of XL cycle PhD research in cybersecurity - CUP E83C24002600001. The views expressed are those of the authors and do not represent the funding institution.

This work was supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP F83C22001690001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART")

Declaration on Generative AI

During the preparation of this work, the author(s) used WRITEFULL in order to perform Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] W. Stallings, Cryptography and Network Security: Principles and Practice, 5th ed., Prentice Hall, 2007. URL: https://www.pearson.com, discusses VPNs in the context of network security.

[2] M. Bishop, Introduction to Computer Security, Addison-Wesley, 2003. URL: https://www.pearson.com, covers authentication protocols and related topics in computer security.

[3] A. A. Elkhail, N. Lachtar, D. Ibdah, R. Aslam, H. Khan, A. Bacha, H. Malik, Seamlessly safeguarding data against ransomware attacks, IEEE Trans. Dependable Secur. Comput. 20 (2023) 1–16. URL: https://doi.org/10.1109/TDSC.2022.3214781. doi:10.1109/TDSC.2022.3214781.

[4] Compliant worm storage using netapp snaplock, https://www.netapp.com/pdf.html?item=/media/6158-tr4526pdf.pdf, 2023.

[5] 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, 2024.

[6] Write-once-read-many (worm) tamper proof technology, https://www.nexusindustrialmemory.com/write-once-read-many/, 2023.

[7] S. Smalley, C. Vance, W. Salamon, Implementing selinux as a linux security module, NAI Labs Report 1 (2001) 139.

[8] Enabling isg to interact with external policy servers, https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/isg/configuration/xe-3s/asr1000/isg-xe-3s-asr1000-book/isg-ext-pol-svrs.pdf, 2017.

[9] Crowd strike 2022 global threat report, https://go.crowdstrike.com/global-threat-report-2022.html, 2022.

[10] Kaspersky security bulletin 2021, https://securelist.com/ksb-2021/, 2021.

[11] A. Continella, A. Guagnelli, G. Zingaro, G. D. Pasquale, A. Barenghi, S. Zanero, F. Maggi, Shieldfs: a self-healing, ransomware-aware filesystem, in: S. Schwab, W. K. Robertson, D. Balzarotti (Eds.), Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016, ACM, 2016, pp. 336–347. URL: http://dl.acm.org/citation.cfm?id=2991110.

[12] J. Huang, J. Xu, X. Xing, P. Liu, M. K. Qureshi, Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware, in: B. Thuraisingham, D. Evans, T. Malkin, D. Xu (Eds.), Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, ACM, 2017, pp. 2231–2244. URL: https://doi.org/10.1145/3133956.3134035. doi:10.1145/3133956.3134035.

[13] P. Caporaso, G. Bianchi, F. Quaglia, Vaultfs: Write-once software support at the file system level against ransomware attacks, arXiv preprint arXiv:2410.21979 (2024).

[14] Fs-verity: read-only file-based authenticity protection, https://docs.kernel.org/filesystems/fsverity.html, 2019.