How to cite this article: *Enrique Soriano-Salvador, Gorka Guardiola-Múzquiz, SealFS: Storage-based tamper-evident logging, Computers & Security, Volume 108, 2021, 102325, ISSN 0167-4048.*

DOI: https://doi.org/10.1016/j.cose.2021.102325.

SealFS: Storage-Based Tamper-Evident Logging

Enrique Soriano-Salvador,Gorka Guardiola-Múzquiz
Universidad Rey Juan Carlos
`enrique.soriano@urjc.es, gorka.guardiola@urjc.es`

Abstract

Log analysis is essential for a forensic investigation. Upon intrusion, log files are usually forged in order to hide or fake evidence. If the system is completely compromised, malicious code can be executed in kernel or hypervisor mode making even signed log files vulnerable. As a countermeasure, some systems archive the log files on another system through the network. This solution is not always suitable or desirable and it just shifts the problem elsewhere. The log files need to be preserved on another networked machine which may itself be attacked. In this paper, we present a simple scheme to authenticate local log files based on a forward integrity model. The scheme is based on a realistic assumption: *nowadays, storage is very cheap*. We can authenticate the logged data generated, starting from boot time to the instant that the malicious code elevates privileges. This tamper-evident scheme does not depend on special security hardware or securing a distributed system. We also present a prototype implementation of this scheme, SealFS. Our implementation, which showcases this approach, is a novel stackable file system for Linux. It enables tamper-evident logging to all existing applications, provides backwards compatibility and instant deployability. Last, we present a performance evaluation of this prototype that shows the viability of this approach.

Keywords: Cybersecurity, logging, file system, tamper-evident, verification, authentication, forensics.

1 Introduction

When a system is compromised, attackers usually try to forge the log files in order to delete or counterfeit logged events that could provide evidence of the attack (e.g. network addresses, vulnerable accounts, attack vectors and so on). Tamper-evident logs are fundamental for digital forensic triage.

Most systems include components and libraries to log system and user events. These mechanisms must be lightweight from two points of view: space (RAM, disk) and time (CPU) [1]. Conventional operating systems such as MS Windows and Linux include this kind of support. Moreover, these components provide confidentiality and integrity of logged data. Nevertheless, it is difficult to provide these guarantees if the attacker is able to get administrator privileges. Once the elevation of privileges happens, the attacker can modify the log files even if they are signed and/or encrypted. As a user with administrator capabilities, the attacker is also able to debug the memory (userspace and kernel) and find any key used to sign or encrypt logs. Of these two properties, confidentiality and integrity, we will focus in integrity. Confidentiality is out of the scope of our work.

The present work is focused on a forward integrity model [2] that assumes that the adversary is able to run code at any privilege level. The main features are:

- **Simplicity.** Some solutions found in the literature are considerably complex [3, 2, 4, 5, 6, 7, 8, 9]. These solutions are old. As other authors explain [10], *it is surprising that commodity operating systems offer no special protections for their logging frameworks.* The intrinsic complexity of these approaches may be the cause. In contrast, we propose a simple scheme based on a realistic assumption: *nowadays, storage is very cheap.* At the time of writing, a 32 GB USB flash drive is less than $5.

- **Local and autonomous operation.** The system is designed for local operation, therefore it is suitable for disconnected, loosely connected and partitionable scenarios. Other approaches handle logs in a distributed fashion, sending the logged data to an independent system (i.e. a logging server) or a cloud based service. In this case, the attacker must control both the target system and the logging server/service to compromise the evidence. Other solutions follow more complex distributed schemes (e.g. based on blockchain technology). The problem is that distributed solutions are not suitable or desirable for many scenarios, for example, autonomous robots operating at isolated areas or disconnected systems. Also, the attacker may cut the connection to the log server, making it look like an accident or an unrelated attack. For example by running a DoS attack on a nearby service and collapsing the network.

- **No need for specialized hardware.** Some other approaches need specialized hardware, for example printers (continuous hard copy printing of the logs), WORM (Write Once Read Many) devices or secure hardware (e.g. TPM [7] or secure enclaves [11, 12, 10]) to provide tamper-evident logs. Non-conventional systems, such as robots, mobile and embedded systems, may be unable to use this kind of hardware.

- **Backwards compatibility, instant deployability and log availability.** Our implementation authenticates the written data at operating system level. Thus, applications and logging components do not need to be aware of the new feature. Unmodified programs can use the standard operating system interface to write to the system logs, as usual. They can also read the logs normally (for inspection, backups, etc.).

- **Good performance for logging and verification.** The system provides fine-grained audits and fast verification for independent log files. In addition, the implementation is lightweight enough to be run on limited machines. The keys used by our system (*keystream*, described later) could also be preserved encrypted in the same machine with the logs. It is after all only a file. It could be protected by a password or a key or any other way. As it is not needed for regular functioning, physically removing it by means of an authentication device is one posible solution which prevents many attacks, but by no means the only one. The authentication device is, by design, sparingly used. Its only purpose is a forensic audit of the logs. It is not unmanageable to keep it inside a secure place, like a safe or in an office under lock and key. The auditor will take it out only when it needs to be used. Again, the trade-offs have been explored extensively in the literature.

- **Password-free** Our system verifies the logs by using a USB portable device (i.e. a *"something you have"* method) in order to avoid passwords. This property is discussed further in 4.4.2.

In our approach, the administrator generates a long random key (which we call keystream) at the time of configuration. The keystream is duplicated in an external storage device that will be physically disconnected from the system (e.g. a USB pen drive). encrypted (e.g. stored on an encrypted file system) and/or signed by third parties (e.g. auditing authorities).

At execution time, the data appended to the logs is authenticated using a portion of the keystream. Then, this portion is *burnt*, i.e. it is deleted from the keystream and the memory. The authentication data generated using this portion is stored in a new log, together with other metadata. To verify the logs, the auditor has to use the duplicated keystream (stored in the external device), the authentication data and the metadata.

We present SealFS as an implementation of this idea. SealFS is a Linux kernel module that implements a stackable file system. When it is mounted on top of another file system, it protects all the files under the mount point. It only allows append-only write operations and authenticates the data written to the underlying files (served by other file system). Appending is the only important mutating operation for a log, so there is not need to support authentication for other operations. Forbidding any other modification of the log improves security without any real loss in functionality.

1.1 Use Case

We can distinguish three phases when a system is attacked:

1. The system is functioning normally, not compromised in any way. In this phase, the attacker probes the system for weaknesses and tries different remote or local exploits depending on the vector of the attack. Hopefully, log files capture information of the kind of attack and the attacker during this phase.

2. The second phase starts when the system is compromised and the attacker is inside the system, but does not have full control. In this phase, the attacker is trying to elevate privileges. The logs may capture some traces of this phase, though it may be easier for the attacker to hide malicious actions.

3. The attacker elevates privileges and is in full control of the machine. The attacker is able to delete and manipulate the logs in order to make the traces of the attack disappear.

When a system administrator is trying to pore over the logs and binaries of a machine, she does not know in what phase of attack the system may be or indeed if there has been an attack. Being able to, either trust the content of the logs or check whether they have been manipulated, can be a valuable and time saving tool. This is what SealFS provides. With SealFS, an attacker in the third phase of an attack will not be able to delete all traces of the attack in the logs. Specifically, the attacker cannot manipulate or forge logs made in the past (first and second phase) without trace. This enables the system administrator to detect it. SealFS means to provide two features:

- A yes or no answer to the question: "Have parts of the log files generated in phases 1 and 2 been deleted or manipulated in any way?".
- If the answer is yes, SealFS lets the system administrator verify which parts of the log files were manipulated and which can still be trusted.

The common use case of SealFS is:

1. The administrator configures SealFS in machine A. To do so, the administrator connects an external USB pen drive and runs a configuration command. A long keystream is created and stored on machine A (e.g. on a conventional disk). It is also copied to the USB pen drive (optionally encrypted and/or signed).
2. Once SealFS is configured, the administrator disconnects the USB pen drive and keeps one or several copies of this device in a safe place.
3. Machine A starts its normal execution. Applications and services append data to the log files normally. Note that SealFS is mounted at boot time by the init process (`systemd` is the most common implementation these days). Note also that the USB pen drive is not needed for normal execution.
4. When the administrator needs to perform the forensic analysis and audit these log files, she mounts machine A's disk in a trusted machine B. Then, the administrator connects the USB pen drive and runs the verification command, which provides the yes or no answer detailed above.
5. If the verification is successful, the administrator looks for evidence in the log entries generated in phases 1 and 2.

Note that while the attacker may forge new logs and this can go undetected, old logs cannot be changed without trace.

1.2 Organization of the Paper

The paper is organized as follows: section 2 discusses the state of the art, section 3 states the threat model, section 4 presents our approach, section 5 describes the implementation of SealFS, section 6 shows its evaluation and section 7 presents conclusions and future work.

2 Related Work

There are too many related works (products and publications) for us to cite them all here. In this section we only highlight a few representative works. Comparing our system with those should be enough to understand the place our system occupies within the literature.

Conventional operating systems implement different schemes to manage logs. Zeng et al. [1] published a survey that describes the most common approaches (conventional logging, WORM devices, signed logs [13], etc.) and the logging scheme of Linux and Windows operating systems. None of the software solutions described in that survey provide tamper-evident logs for the threat model explained in section 3.

Zeng et al. [14] also proposed a non-cryptographic model for accounting. In this model, all system administrators can be accounted for, even if they are untrustworthy. It adds new user policies that are implemented in the kernel, which must be trusted. In our threat model, the kernel is not trusted after the exploitation. Moreover, any malicious code running in kernel or hypervisor mode can access the storage devices directly (or the memory) and change the logs. Last, this approach may require deep changes in the system's user model (affecting backwards compatibility).

Integrity checkers like I^{3}FS [15] checksum some files (e.g. binaries) and define policies of what to do when the files do not match. Again, I^{3}FS does not protect against our threat model. The attacker only needs to isolate the network (or do it while the system is disconnected or out of range if it is a robot), elevate privileges, rewrite the logs and regenerate the checksums. It does not provide forward integrity if the attacker can run kernel code (by elevating privileges). Its implementation is similar to ours in that it is a stackable file system.

Other solutions are based on virtualization (jails/containers and VMs), for example the one proposed by Chou et al. [16]. Again, these approches do not work if the attacker is able to run code in hypervisor mode or break isolation.

There are many distributed approaches and commercial products for remote logging. The common solution is to follow a client/server scheme to send the logs to a remote server (Linux's *syslog* and Windows' *event log* can do that) or the cloud (see for example [17, 18]). Some systems based on history trees and randomised binary search trees (hash treaps) [19, 20] also depend on remote servers. Self-securing storage like S4 [21] separates storage moving it to a different system providing a RPC interface. This second storage system is simpler and better audited and logs all interactions. This audit log is exported read-only to the RPC interface.

As stated in the introduction, distributed solutions are not suitable for disconnected or loosely coupled systems (e.g. autonomous robots). Moreover, sending the logs to a remote server does not solve the problem because the server can be attacked to remove evidence of the attack. Note that logging servers can also take advantage of our approach to protect the logs of their clients.

Recently, several distributed schemes and products based on blockchain have been proposed [22, 23, 24, 25]. These schemes are intrinsically distributed, so they are not viable for disconnected scenarios.

More closely related to our work, several authors have proposed the use of cryptographic ratchets [3, 26] (e.g. hash chains or linear Merkle trees) to generate the keystream to authenticate the logs and provide forward integrity. After the exploitation, the adversary is not able to reverse the ratchet and recover the keys used to authenticate the logs. The only way to regenerate a ratchet is by using the key (i.e. the seed) and it is not stored in the system. The administrator must provide the key when the system is configured and when the logs are verified. This approach was proposed a long time ago [2, 4, 5, 6]. Logcrypt [27] evolved the scheme to support concurrent logging and provide third-party verifiability through public key cryptography. Section 4.4.1 further compares our approach and ratchets.

Ma et al. proposed a scheme to provide *forward-secure stream integrity* [8] that combines the authentication tags (signatures or MACs) to generate a single aggregate tag. Therefore, it is more efficient in terms of storage. Yavuz et al. proposed similar approaches [9, 28]. Hartung et al. also proposed an scheme to aggregate signatures [29]. Our proposal is based on the assumption that current storage devices are huge and cheap, so storing keystreams and integrity metadata is not a problem. Moreover, some of these systems have security issues [30].

Sinha et al. [7] also evolved the ratchet approach to address some of the issues of previous proposals. In addition, this work uses secure hardware (TPM 2.0) to store the keys and also uses a monotonic counter. Our approach provides the same advantages as this proposal: disconnected operation, autonomous reboot, log rotation, modular verification and good enough performance for tamper-evident logging without depending on specialized secure hardware. Moreover, it can be integrated in a regular Linux kernel.

There are other proposals that use secure enclaves with sealing and unsealing primitives to ensure integrity and confidentiality of log data [11, 12, 10]. These systems depend on specialized hardware. For example, SGX-log [12] depends on Intel SGX hardware. Another recent example is CUSTOS [10]. This system is based on a tamper-evident logger and a decentralized auditing protocol. The tamper-evident logger runs on a TEE (Trusted Execution Environment) so it requires it to protect its code. This kind of hardware has been subverted in the past see, for example, [31]. The presented prototype also depends on Intel SGX [10]. As stated before, SealFS does not require any specialized hardware to provide tamper-evident logging.

There are multiple security schemes that depend on portable devices like USB, NFC and Bluetooth tokens based on the U2F protocol [32, 33, 34], smart cards, etc. These devices are normally used to provide single or multi factor authentication. As far as we know, no other system uses a portable device to provide tamper-evident logging.

3 Threat Model

The asset is the logging data generated by the the applications, which will be used to perform a forensic analysis.

We focus on the *forward integrity model* [2]. It ensures that the logs generated before the exploitation are tamper-evident. In this case, we focus on a strong variant of this model: once the system has been compromised (phase 3), the adversary controls the whole system (software and hardware).

3.1 Threats

- The attack may be local or remote. The attacker may have physical access to the hardware.
- Upon exploitation, the adversary is able to execute code in any privilege level (from Ring 3 to Ring[1] -3) and control all the hardware.
- The attacker can delete or modify any data created before or after the attack.

[1]We are using here the commonplace conventions where: Ring -1 is hypervisor mode, Ring -2 is the SMM (system management mode) and Ring -3 is the Management Engine.

3.2 Dependencies

There is only one dependency: secondary internal and external storage devices must be big enough to store a keystream for long time operation. The system's hardware is standard (e.g. a conventional Intel or ARM based computer with common storage devices).

3.3 Assumptions

- The system may be online or disconnected. The software and hardware are working normally and are trustworthy until the adversary exploits the system, corrupts its processes or executes malicious code.
- The cryptographic algorithms used by the system are correct. We assume that the HMAC algorithm [35] used by our approach is secure: the attacker cannot forge authentication data or recover the keys used to generate it.
- The attacker is not able to deactivate or bypass our system in phases 1 and 2 (administration privileges are required to do that) but is able to do anything in phase 3.
- The deletion procedure used by the system is secure. Once data is deleted from a device, it cannot be recovered (from memory, cache or disk). This is arguably the weakest point of the system. It is difficult with modern hardware to guarantee safe deletion of data in a hard disk, but mitigation procedures can be put in place to deter but the most-resourced adversaries, see section 5.1.3 for more details.
- The auditor verifies the logs in an independent, correct and trustworthy system that mounts (as read-only) the original storage device or a copy. The hardware used by the auditor is standard.

3.4 Mitigation

The auditor can verify if the log entries **generated in phases 1 and 2** have been tampered with. If the attacker modifies or removes any logging data generated in phases 1 and 2 (or the corresponding metadata), the auditor can detect it. Note that the attacker may forge new logs in an undetectable way in phase 3 of the attack. By that phase, the unburnt parts of the keystream are available to the attacker. Past logs, though, are still protected, the secrets to manipulate them are not available any more.

Note also that, even if the system is compromised, the logging data may still be correct (that is, the attack may not have manipulated the logs). In this case, the auditor can state that the system is compromised only if the logs provide evidence about the exploitation.

4 Our approach

As stated before, our approach is simple. It uses two different storage devices: α and β. β is an external storage device that will be used to verify the logs in the future.

4.1 Configuration

1. A random keystream K is generated.
2. K is stored in both devices (in a file in a binary format which includes a header). K_α is the keystream stored in α and K_β is the keystream stored in β.
3. K_α and K_β headers are initialized. They have two fields: $K_\alpha header.id$ is an id number for the keystream which both K_α and K_β will share to pair them; $K_\alpha header.off$ is the current offset of the keystream, which is initialized to zero on both α and β.
4. β is physically disconnected from the system and kept in a safe place.
5. The system creates a file, $SEAL_{log}$, to store the authentication data and metadata for the logs.

4.2 Writing the log files

$HMAC(key, msg)$ is a secure keyed-hash message authentication code algorithm [35]. The input of the function is a key and a message. The output is a secure digest of the message that depends on the key.

Only append write operations are allowed (i.e. write operations at the end of the log). When some data D_i of size Dsz_i has to be appended to a log file L at offset $Loff_i$, the following operations are executed:

1. The data is appended to the log.
2. The current offset of K_α is read from its header.

3. A chunk (C_i) of K_α is read. The chunk size, which determines the length of key consumed per write is constant (Csz) and independent of the size of the write to the log.
4. The corresponding zone of K_α is *burnt*.
5. The updated offset of K_α is written to its header.
6. The HMAC of the concatenation of the log id (L), which identifies uniquely the log file, the offset in the log ($Loff_i$), the data length (Dsz_i), the offset in K_α for C_i ($Coff_i$) and the data (D_i) is calculated. The chunk C_i is used as the key.
7. The chunk is removed from memory.
8. A record R with fields $L, Loff_i, Dsz_i, Coff_i$ and the HMAC is created.
9. The record R is appended to $SEAL_{log}$.

Algorithm 1 Write algorithm

```
1: append $D_i$ to $L$ at offset $Loff_i$
2: $Coff_i \leftarrow K_\alpha header.off$
3: $C_i \leftarrow K_\alpha[Coff_i \ldots Coff_i + Csz - 1]$
4: $K_\alpha[Coff_i \ldots Coff_i + Csz - 1] \leftarrow RANDOM()$
5: $K_\alpha header.off \leftarrow Coff_i + Csz$
6: $H_i \leftarrow HMAC(C_i, L||Loff_i||Dsz_i||Coff_i||D_i)$
7: remove $C_i$ from memory
8: $R \leftarrow (L, Loff_i, Dsz_i, Coff_i, H_i)$
9: append $R$ to $SEAL_{log}$
```

Operations 2-9 must be executed atomically to preserve the integrity of $K_\alpha header.off$ and the order of $SEAL_{log}$. This is from the perspective of the concurrency of the algorithm. If this invariant failed to be preserved, it would still be detected in an audit.

The system needs extra space to store four integer values and an HMAC digest per write operation (independently of the number of bytes written to the log file by each append operation).

4.3 Verification

When the auditor needs to verify a log L, to see if it has been manipulated, she has to attach the β device and execute algorithm 2.

First, the keystreams are checked: their size and id numbers must match and the *burnt* area must end at $K_\alpha header.off$. Then, all the records of $SEAL_{log}$ are verified sequentially. Note that:

- The keystream is burnt sequentially. The keystream preceding $K_\alpha header.off$ has to differ between k_α and k_β and be the same (unburnt) after.
- Given two contiguous records, their corresponding chunks must be contiguous. O_K is used to check that.
- Records belonging to a log L are ordered in $SEAL_{log}$ (by $R.Loff$). Thus, the data areas defined by its records must be contiguous: if there is a gap, the verification fails. An array with a position for each log file, $O[]$, is used to check that.

The corresponding chunk of the keystream is read from K_β (at offset $R.Coff$), the data described by the record is read from L (from position $R.Loff$, length $R.Dsz$) and the HMAC is regenerated using the chunk as the key. The new HMAC and the HMAC stored in R are compared. If they are not equal, the verification fails for L.

TODO, check in verification algo, the log is sealed (number of entries multiple of NRATCHET, like implementation).

Provided that the HMAC is secure, the attacker is not able to deduce the key (C) for any record stored in $SEAL_{log}$. Thus, she will not be able to forge any HMAC of $SEAL_{log}$.

If the adversary removes any record for a log L from $SEAL_{log}$, the verification fails (line 14). If any log file is truncated or shortened, it also fails (line 14). If the adversary modifies any of the fields of any record belonging to L or its data, the verification fails because the HMAC does not match (line 21).

Modular log verification (i.e. verifying only a portion of L) would be done similarly. In this case, we would check only the records for the corresponding area of that specific log file. Note that, in this case, we can only claim that the checked area of this log file has not been tampered with. Other areas of this log file (or any other log file) could be tampered with and the file could be truncated.

4.4 Discussion

4.4.1 SealFS vs. Ratchets

The main advantage of our approach is its simplicity. Ratchet based proposals have been around for more than 20 years but even though they have

Algorithm 2 Verification algorithm

```
1: if $size(K_\alpha) \neq size(K_\beta)$ or $K_\alpha header.id \neq K_\beta header.id$ then
2:     FAIL()
3: end if
4: $P \leftarrow K_\alpha header.off$
5: if $K_\alpha[P-Csz \dots P-1] = K_\beta[P-Csz \dots P-1]$ or $K_\alpha[P \dots P+Csz-1] \neq$
   $K_\beta[P \dots P + Csz - 1]$ then
6:     FAIL()
7: end if
8: $O_K \leftarrow 0$
9: for each record $R$ of $SEAL_{log}$ do
10:     if $O[R.L]$ is not initialized yet then
11:         $O[R.L] \leftarrow R.Loff$
12:     end if
13:     if $O[R.L] \neq R.Loff$ or $O_K \neq R.Coff$ then
14:         FAIL()
15:     end if
16:     $C\prime \leftarrow K_\beta[O_K \dots O_K + Csz - 1]$
17:     $MD \leftarrow R.L||R.Loff||R.Dsz||R.Coff$
18:     $D \leftarrow R.L[R.Loff \dots R.Loff + R.Dsz - 1]$
19:     $H\prime = HMAC(C\prime, MD||D)$
20:     if $H\prime \neq R.H$ then
21:         FAIL()
22:     end if
23:     $O[R.L] \leftarrow O[R.L] + R.Dsz$
24:     $O_K \leftarrow O_K + Csz$
25: end for
26: SUCCESS()
```

found specialized use they have not been integrated in conventional operating systems yet.

Note that the ratchet, similarly to SealFS, starts with a number of secrets (the seed, a number of keys or some secret state). On each "epoch" (which would correspond to a write in our system) it is advanced with a non reversible function and then saves its status in permanent storage (NVRAM, disk or specific purpose storage hardware). This is necessary in order to support reboots, autonomous or not. Moreover, the ratchet also needs to update other metadata in permanent storage (a file analogous to $SEAL_{log}$). Thus, a ratchet implementation has to pay the same price: atomically updating some metadata located in permanent storage which includes deleting old state securely and keeping a secret in separate storage for later verification.

There is a more fundamental difference from an theoretical point of view. As stated before, a ratchet is a (degenerate, linear) Merkle tree [36] which uses an HMAC instead of a regular secure hash. An HMAC is a keyed secure hash. The output of each HMAC is used as the key to compute the HMAC of the logs, which will be verified later. In the ratchet, on each "epoch" the keys used to compute the HMACs are obtained from the previous secret state (and maybe some state chained from the output of the previous HMAC of the log).

As a consequence, the total entropy of the system is limited to the size of the initial secret. The original paper [2] itself states that *"the security of the prf-chain FI-MAC degrades only linearly with the number of epochs"*. The number of epochs would correspond to the number of writes in our system. In contrast, SealFS adds new entropy per input record in the log and does not connect the secrets in a Merkle tree. Burning the keys already provides us with the necessary forgetful (non-reversible) process. This makes it theoretically more secure even if it uses more disk space. Also, as a consequence, our scheme provides random access to the keystream in verification time; we do not need to calculate the keys, they are already there. Ratchet based keystreams must be regenerated from scratch to verify the logs (even to verify a small part of a single log file). This can be a problem for large log files.

In general, ratchets need to store keys or seeds. They can be kept on an external device or locally. If they are stored locally, they must be protected with secrets (i.e. passwords), which carry their own set of security issues. On the other hand, if they are stored on an external device, they should be kept well protected and separate from the main system for security reasons. In this case, the only difference with our approach is that we will use the whole storage capacity of the external device. The cost per byte of storage media has dropped exponentially for three decades [37]. Nowadays, storage is several orders of magnitude cheaper and faster than when ratchets first appeared, so we can explore new, simpler approaches that can work well in limited CPUs like those used in robots and IoT devices. At the time of writing, it is difficult to find an external USB device smaller than 32 Gb. All the extra space will be unused in the ratchet scheme. Note that any approach used by ratchets to store their secrets (e.g. a password protected encryption) can be used to protect K_β, following a two-factor method (*"something you have"* and *"something you know"*) if required. We just need to decrypt K_β while performing the audit.

It is also interesting to notice that future approaches can combine both ideas, for example, burning a chunk of the keystream per new file created and using a ratchet with that key as input. Our approach is closer to the extreme of the spectrum, the one-time pad (OTP) [38] approach. As a consequence, it consumes a space proportional to the length of the message (actually in our case proportional to the number of writes to the log, which is much better) and provides more security whereas the ratchet is at the other extreme of constant CPU usage and linearly degrading security and verification time. Combining both can enable the user to choose the adequate compromises for their use case.

We also agree with the authors of the original ratchet paper [2] that *"Other than a one-time pad, the authors know of no cryptographic primitives with forward privacy"*. Note that directly using a one-time pad with the Vernam cipher (an *xor* with the logs or some hash of the *xor* with the logs) is not feasible. Some parts of the content of the logs is known beforehand. As a consequence, the process could be reversed to recover the corresponding section of the secret and partially forge the logs. Our scheme is probably the closest thing which is actually feasible. Another problem of a one-time pad would be that its secrets are equal to the size of the message (the logs in this case). In our scheme, they are proportional to the size of a write. If each write is aproximately a line of around 80 bytes and a key is 20 bytes, this is a factor of 4 of reduction of space, of course with a similar cost in the theoretical maximum security which can be provided even with an ideal HMAC function.

4.4.2 Password-free

Our system verifies the logs by using a USB portable device in order to avoid passwords. The bibliography on the advantages and trade-offs of the different authentication methods is vast, see, for example [39, 40, 41, 42, 43]. We delegate the management of the authentication device to the administrator, a specialized user that understands these trade-offs (devices can be lost or damaged, passwords can be forgotten, misused, reused, etc.). The secret keystream we generate, could be kept in a specialized anti-tamper device, should the need for a higher level of security appear, foregoing the easiness of a commonplace and cheap USB device. If keeping the USB device protected is too much of a burden, the keystream could be encrypted with a password, yielding this property.

4.4.3 Whiteouts

We have deliberately not added whiteouts to the log (i.e. entries which log the deletion of a file). It somehow goes against the philosophy of accountability of the system. In order to be able to audit the system completely, it is vital that no information is lost. Storage is big enough already (and keeps growing). It should not be a problem to keep all the logs produced between two audits.

Eventually, the files will need to be audited and, at that time, old files can be deleted or backed up as needed. Take into account that if we added whiteout entries in the log and permited verifiable deletion, it would not be possible to provide protection against our threat model. This would let the intruder delete traces of the attack after elevating privileges. In our threat model, future operations should not be able to delete information of the past.

5 SealFS

SealFS is a stackable file system for Linux that implements the scheme presented in the previous section. It is based on WrapFS [44].

A file system is an operating system component[2] that provides files and directories. Userspace programs access files through the operating system interface (POSIX API): the processes perform system calls to manipulate file data and metadata.

A stackable file system is a file system that is mounted on top of another one to offer extra functionality (e.g. directory union, encryption, etc.).

SealFS is mounted on top of a directory that has the logs we want to protect. It serves the same files and directories as the underlying directory. Its goal is to handle the write operations that appends data to an underlying file, applying the proposed scheme. Note that, in the end, the data is written in the underlying file (the one served by the underlying file system), but it is authenticated on the fly.

For example:

[2]Note that some file systems run in userspace (e.g. FUSE based file systems), but file systems are commonly implemented as a kernel space component.

```
$>  logdir=/var/log/antitamper
$>  stat -f -c '%T' $logdir
ext3/ext4
$>  ls -l $logdir
total 0
-rw-r--r-- 1 root root 0 Apr 10 14:21 robot.log
-rw-r--r-- 1 root root 0 Apr 10 14:21 sensors.log
$>  mount -t sealfs   $logdir $logdir -o kpath=/kalpha
$>  ls -l  $logdir
total 0
-rw-r--r-- 1 root root 0 Apr 10 14:21 robot.log
-rw-r--r-- 1 root root 0 Apr 10 14:21 sensors.log
$>  mount | fgrep  $logdir
/var/log/antitamper on /var/log/antitamper type sealfs (rw)
$>
```

First, the example shows that `/var/log/antitamper` is a directory server by an Ext4 file system[3]. The directory is listed and there are two empty files. Then, SealFS is mounted on `/var/log/antitamper`. The path to the keystream (K_α) is provided as a mount option (`/kalpha`). From now on, when an application opens any file or directory under `/var/log/antitamper`, it will be using SealFS. All write operations will pass through to the SealFS instance. Next, the `/var/log/antitamper` directory is listed again: it has the same files names as the first listing. Last, the example shows that SealFS is mounted in the corresponding point.

Figure 1 depicts the regular use (the application is writing to the log file served by a standard Linux file system) and the case configured in the previous example.

SealFS will hook the file operations performed by the application to authenticate the written data. When it is mounted, only append write operations are allowed. The file system also forbids many operations not suited for a log file: deletion, memory mapping (`mmap`), etc. It permits to create new files and rename the old ones in order to rotate the logs.

Applications write logs as usual. They open the file and perform append write operations. Multiple processes or threads can write the log files concurrently. Different applications could use independent instances of SealFS mounted in separate points. Once the applications are stopped and SealFS is unmounted, the directory is available through the original file system (i.e. the Ext4 file system in this example): the files (the logs and the metadata file) are regular files.

SealFS will be mounted as part of the boot process. From the moment the system boots, any of the logs in the mount path (for example `/var/log`) will

[3]Ext4 is the standard Linux file system.

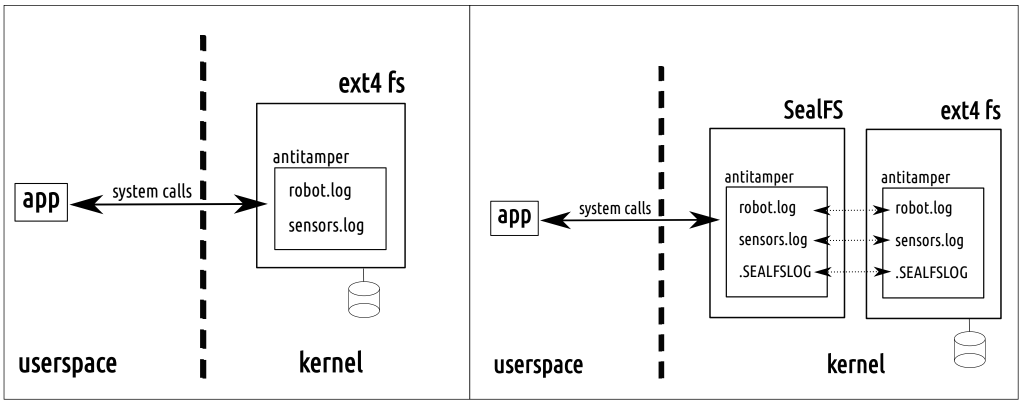

Figure 1: The regular use (left) and the scenario configured in the example (right).

have anti-tamper guaranties. This approach provides instant deployability and backwards compatibility: applications, libraries and frameworks do not require any modification to generate tamper-evident logs. Moreover, our approach is compatible with all standard Unix tools that work with regular files.

5.1 Implementation details

Our prototype of SealFS is fully functional and its implementation is not trivial. In general, kernel code is complex and the implementation of a file system is not straightforward. Note that in the following description the only data structures to support the SealFS file system in secondary storage (the hard disk) are the log files and the SealFS log file (i.e. SealFS does not need a partition). The file system is a software construct and the only “real” file system is the one stacked under it, containing the log files being protected. Next, we summarize the most important features and issues.

5.1.1 The File System

VFS (Virtual File System, also known as the Virtual Filesystem Switch), the Linux subsystem that provides an abstraction layer to implement file systems, is based on the concepts of Unix file systems: the *superblock*, *i-nodes* and *directory entries* which will be data structures in memory.

In Unix-like systems, the main data structure of a file system is the *superblock*. This data structure contains the high-level metadata of the file system. The superblock of a SealFS instance contains:

- A reference to the superblock of the underlying file system.
- The references to read and write the files that contain the authentication metadata for the log files ($SEAL_{log}$) and the keystream (K_α). The paths of these files are specified in the mount operation options. By default, $SEAL_{log}$ is a *hidden* file named `.SEALFSLOG` in the mount point and both files are opened to perform asynchronous write operations
- The data structures used to generate the HMACs are included in the superblock in order to avoid unnecessary allocations/deallocations. The current implementation uses the HMAC-SHA-1 [35] algorithm to authenticate the logging data[4]. The key length used for the algorithm is 20 bytes.
- The data structure used for concurrency synchronization.

Note again that, in this case, the superblock is not stored in any physical storage device: it exists only in memory. It is created when the file system is instantiated and initialized (that is, when it is mounted) and removed when the file system is unmounted.

Applications must open their log files after SealFS is mounted. If the log files are opened by an application before mounting SealFS, the process is accessing the underlying file system directly.[5] In this case, the data written will not be authenticated and the subsequent verification will fail. As an extra protection, SealFS checks if the files of the directory are being used by any process during its initialization. If so, the mount operation fails. It is just a check, but it does not ensure that there are not processes accessing the files bypassing SealFS. A process could open a file while SealFS is being mounted. Moreover, SealFS cannot prevent a process in a different name space from accessing these files through the underlying file system. In conclusion, the administrator must ensure that the applications open the logs after SealFS is mounted (by mounting it at boot time).

In Unix-like systems, a file system element (a file, directory, symbolic link, fifo…) is identified by its *i-node* number. The i-node number is used by the file system to locate the i-node structure of the element. This per-element structure contains its metadata (modification date, permissions, etc.) and

[4]SHA-1 is deprecated, but HMAC-SHA-1 is considered secure [45].

[5]Note that (i) unmounting SealFS requires administration privileges and (ii) SealFS should be mounted by the init process at boot time. An attacker with enough privileges may unmount the file system and forge new logs, but the ones created before the attack will still be protected. The attacker cannot access that part of the keystream, it is already burnt.

the references to find all its data blocks in the storage device. Directories are just lists of directory entries that map file names to i-node numbers.

SealFS provides exactly the same files and directories as the underlying file system. It uses the same i-node numbers and names in directory entries.

The metadata generated and stored in $SEAL_{log}$ uses i-node numbers to identify the log files. This way, files can be renamed to *rotate* the logs: renaming a file only changes the directory entry, the i-node number remains the same. Thus, the records of $SEAL_{log}$ are still coherent after renaming the old log and creating the new one.

A rotated log file could be deleted and then ignored as part of the verification process (doing only a partial verification during the audit). We do not recommend this approach because it weakens the security model.

5.1.2 Synchronization

When a `write` system call is performed over a file, the corresponding hook function of SealFS is called. This function performs the write operation on the underlying file.

The main problem is to know the real offset for the append-only write operation. The hook function receives as a parameter the current offset managed by the file descriptor (i.e. what the process believes the offset of the file is). Nevertheless, this value is ignored by the file system for a write operation when the file is open in append-only mode: the data is always appended at the end of the file, regardless. Moreover, different processes may have different file descriptors for the same file.

SealFS performs the write in the underlying file and when it is completed, the actual number of bytes written to the file is known. Then, SealFS synchronizes the i-nodes (that is, it copies the fields of the underlying i-node to its i-node). The metadata stored in the i-node includes the current size of the file. Then, it calculates the real offset for this write operation (current size minus the number of bytes returned by the write operation). In order to avoid race conditions (e.g. when two processes write the same file concurrently), the process must acquire a per-file mutex to execute these operations.

This mutex is stored in the corresponding directory entry memory data structure of SealFS for the file. Write operations over different files can be executed concurrently.

We need a global mutex to preserve the order of operations in $SEAL_{log}$ and the offsets in the header of K_α. As stated before, this global mutex is stored in the superblock memory data structure of SealFS. After the append-only write is done and the real offset is known, the process acquires the global mutex to execute operations 2-9 of algorithm 1. Note that this is costly and can be greatly optimized. For now, we have aimed for simplicity and correctness.

As explained in section 4.2, operations 2-9 of algorithm 1 must be executed atomically. If there is a system crash (e.g. kernel panic, power failure, etc.) while the data or metadata is not already written in the disk, the logs will be corrupted after rebooting. Note that modern operating systems use asynchronous I/O mechanisms and a memory cache for file systems. These mechanisms have a big impact on performance, but they make it worse in the case of a crash. Commonly, the underlying file system (e.g. Ext4) will implement mechanisms to provide data and metadata integrity in the case of crashes (e.g. journaling and soft updates). Anyway, there is a window of time when a crash will corrupt the logs. SealFS provides a mount option (`syncio`) to force synchronous I/O when $SEAL_{log}$ and K_α are written (`O_SYNC` flag for the `open` system call). In this case, by the time the write system calls returns, the output data and metadata are committed to the underlying storage hardware. This option reduces such window of time, but it has a dramatic impact on performance as we will show in section 6.

In any case, after a system crash, the logs must be verified at boot time. If the logs can be verified, the system will boot normally, mount SealFS and run the applications. If the verification fails, the administrator must audit the system and reconfigure SealFS before starting the applications. In this case, the crash is indistinguishable from an attack: we cannot state that the logs are correct.

5.1.3 Tools

Reliably erasing data from the disk may be tricky depending on the type of storage device.

The first write to burn a used portion of the keystream is done synchronously in the context of the write operation performed by the client. After that first write, it is considerably difficult to recover the burnt portion without a specialized lower layer attack.

Asynchronously, a system daemon (i.e. a service) named `sealfsd` is in charge of reliably *re-burning* K_α in background. This daemon reads the offset ($K_\alpha header.off$) periodically and applies the appropriate deletion technique. Note that the blocks being written by `sealfsd` are not accessed by the driver anymore. Therefore, the interference is minimal (just exchanging an integer, the offset). The extra bandwidth required by `sealfsd` is negligible compared to the normal use of the disk[6].

[6]Note that a disk block holds 25 keys in the current implementation. Re-burning the

In theory, it is possible to perform a complex physical analysis to recover information from magnetic disks, even after multiple overwrite operations [46]. In practice, for current high density magnetic disks, it suffices from 3 to 7 write passes[7] (interleaving constant and pseudorandom data) [47]. On the other hand, overwriting the entire visible address space of a solid state disk (SSD) may not be secure enough [48]. In this case, the daemon must use specific utilities provided by the manufacturer. This task could be relegated in the future to an stackable safe deletion file system like restFS [49] or PurgeFS [50].

There are other userspace tools for SealFS:

- `prep` creates the keystreams K_α and K_β by reading data from the secure random data generator of the system. After the execution of this command, the administrator must detach β from the system.
- `dump` lists all the records of $SEAL_{log}$ in a human-readable format. Note that the records are kept in the endianness of the machine.
- `verify` implements the algorithms described in sections 4.3 to verify $SEAL_{log}$. This command receives the following arguments: the directory that contains the logs, the path for the keystreams K_α and K_β, and optionally, the name of the $SEAL_{log}$ file (if this argument is not provided, it uses the default name). It also accepts a i-node number and a range (begin and end offsets) for partial log verification. In this case, it only verifies the records corresponding to that portion of the log file.

6 Evaluation

In this section we will describe a number of experiments performed to analyze the performance of the prototype implementation of SealFS. The goal of the experiments is to measure the impact of the SealFS layer for performing append operations over a standard Linux file system. We have followed a twofold approach.

First we have measured the performance using the standard benchmark, *filebench* [51] with a custom load which creates a set of files, N processes and M threads and makes them append to the files concurrently. To delve deeper in understanding how the file system behaves, we also implemented a custom

last portions of keystream only requires rewriting a few blocks.

[7]The de facto standard wiper on Linux systems, `shred`, performs three pseudorandom rounds by default.

minibenchmark which measures the same thing but gives back more detailed results.

We have measured only append operations. Other file operations are not of interest because they are either just forwarded to the underlying file system (e.g. read operations) or forbidden (e.g. memory mapping or non-append write operations).

All the experiments were measured in a limited machine in order to be representative of different kinds of systems (servers, robots or embedded systems). The machine has a 2.13 GHz Intel Xeon E5506 CPU with four cores, 4 GiB of RAM and two disks: a WDC WD1602ABKS-1 (a standard SATA HDD disk) and a Kingston SKC300S (a standard SATA SSD disk). It runs a Ubuntu Server 18.04 with a custom Linux 4.8.17 SMT kernel. Both disks are formatted with Ext4 file systems, with the default options.

6.1 Filebench

We ran *filebench* to measure how the system would behave in a typical system. We wrote a small program and ran it in various systems to measure the average line length and standard deviation for a line in the standard log files of the system. In the worst case for performance, a log will receive a write per line (in other cases, they may be compressed and batched). Our program reported that the average line length was around 42 characters with a standard deviation of around 40 characters. Armed with this knowledge, we configured *filebench* with the following parameters:

- Number of standard *filebench* concurrent processes and threads writing to the logs[8]. This will simulate both concurrent applications sharing the same sealfs mount (not strictly necessary as there can be a SealFs mount per application) and a multithreaded application sharing a log file.
- 100 byte write operations, which should cover the biggest write in most cases.
- Disk type: the conventional electromechanical rotating disk drive (NOSSD) or the solid-state (SSD) disk described before.

We ran the tests against a naked Ext4 file system and the same file system with SealFs mounted over it.

The results of the measurements can be seen in figures 2, 3, 4, 5, 6 and numerically in tables 1 and 2. Note that the effect of using an SSD disk is

[8]They are independent, both are created with `fork`.

N. Proc.	Ext4 SSD ms	SealFs SSD ms	Ext4 NOSSD ms	SealFs NOSSD ms
1	0.003	0.012	0.003	0.013
2	0.003	0.019	0.004	0.019
4	0.003	0.039	0.004	0.040
8	0.007	0.079	0.007	0.083
16	0.013	0.160	0.015	0.170
32	0.026	0.319	0.031	0.334
64	0.052	0.645	0.069	0.677

Table 1: Filebench measurements for different number of processes. The table shows the average latency of 100 byte write operations.

N. Th.	Ext4 SSD ms	SealFs SSD ms	Ext4 NOSSD ms	SealFs NOSSD ms
1	0.003	0.012	0.003	0.013
2	0.003	0.018	0.004	0.020
4	0.007	0.038	0.007	0.040
8	0.015	0.080	0.016	0.084
16	0.032	0.159	0.032	0.168
32	0.063	0.319	0.066	0.337
64	0.131	0.639	0.137	0.676

Table 2: Filebench measurements for different number of threads. The table shows the average latency of 100 byte write operations.

negligible because of the file system cache. Looking at the measurements, SealFS is, at worst, 10 times slower than Ext4. This makes it, as it is, useable. Most applications do not have logging as bottleneck and will be faster than this worst case, by batching the writes and having a dedicated process for writing to the file. Note that with less concurrency it is just 5 times slower instead of 10.

As can be expected, separating log files in different mount points (i.e. two independent instances of SealFS) makes write operations faster. Note that different SealFS instances do not share any internal resource (files, data structures, locks, etc.). Figure 6 compares the times of the benchmark for one and two concurrent SealFS instances. As the number of concurrent processes increases, the difference decreases, because they share the rest of system's resources (i.e. the contention for the usage of the system resources increases). For example, for two processes, the result is 0.019 vs. 0.013 ms per operation. For four processes, it is 0.039 vs. 0.019 ms per operation. These times match the results shown in table 1.

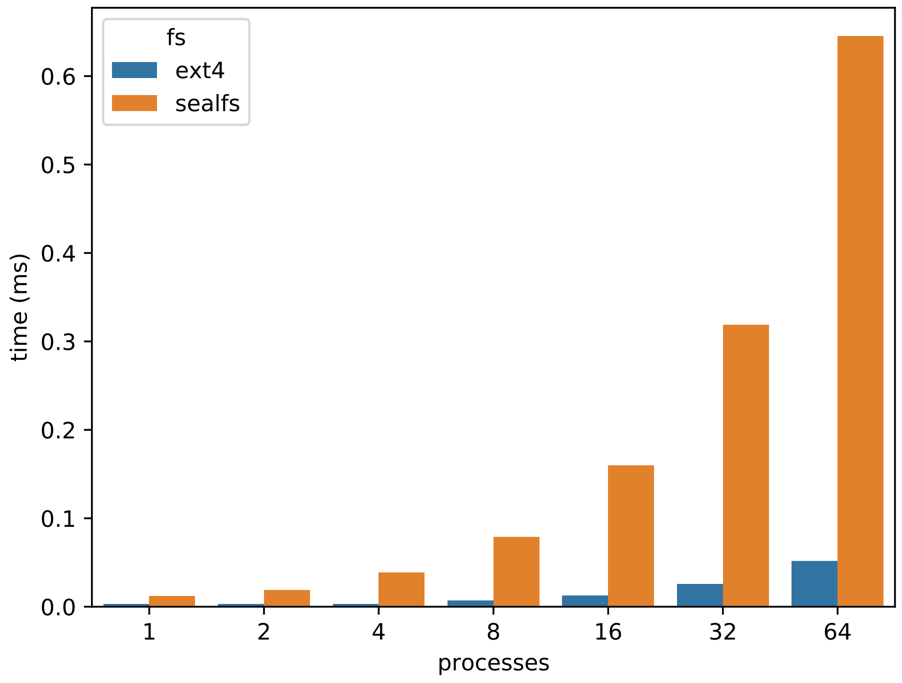

Figure 2: Filebench reported time for 100 byte write operations, SSD disk and private log files.

6.2 Minibenchmark

As we stated earlier, we also wrote a minibenchmark to understand the behavior more in depth for other cases. The custom minibenchmark performs 1000 write operations (per process) from a user space program. We measure the time to complete each `write` system call. We propose a set of scenarios that we consider complete and representative. The variables are:

- Size of each of the write operations (100, 1000 and 10000 bytes).
- Number or concurrent processes writing to the logs[9].
- Number of logs: a log file shared by all processes or a private log file per process.
- Disk type: a conventional electromechanical rotating disk drive (NOSSD) or a solid-state (SSD) disk.

[9]They are independent, standard processes created with `fork`.

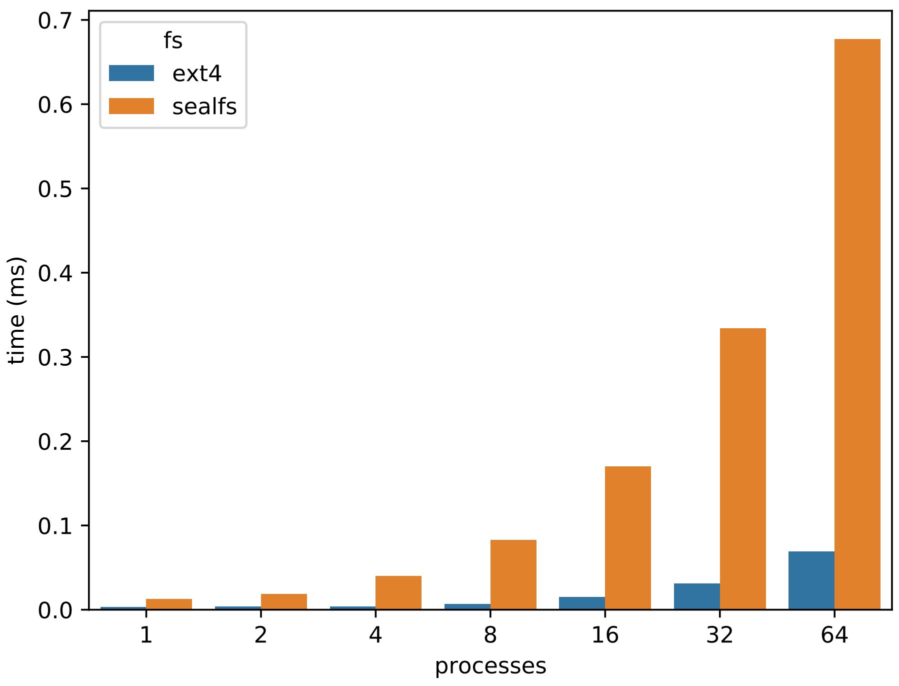

Figure 3: Filebench reported time for 100 byte write operations for concurrent processes, NOSSD disk and private log files.

- I/O type: synchronous or asynchronous write operations for $SEAL_{log}$ and K_α.

6.2.1 Execution

To measure the time to complete a write system call, we read the time stamp counter register (TSC) of the cores from the user space process. The TSC is read before performing the system call and when the system call returns. Then we calculate the number of cycles required for the operation. Note that using the TSC to measure is not trivial: the TSC of the different cores must be synchronized, the rate must be constant, power saving must not interfere, and so on [52]. The user space program reads the TSC following the guidelines published by Intel to benchmark code execution [53], taking into account the *out-of-order* dynamic execution of instructions performed by modern CPUs (using barriers before reading the TSC).

All executions use the same K_α file and write exactly the same data.

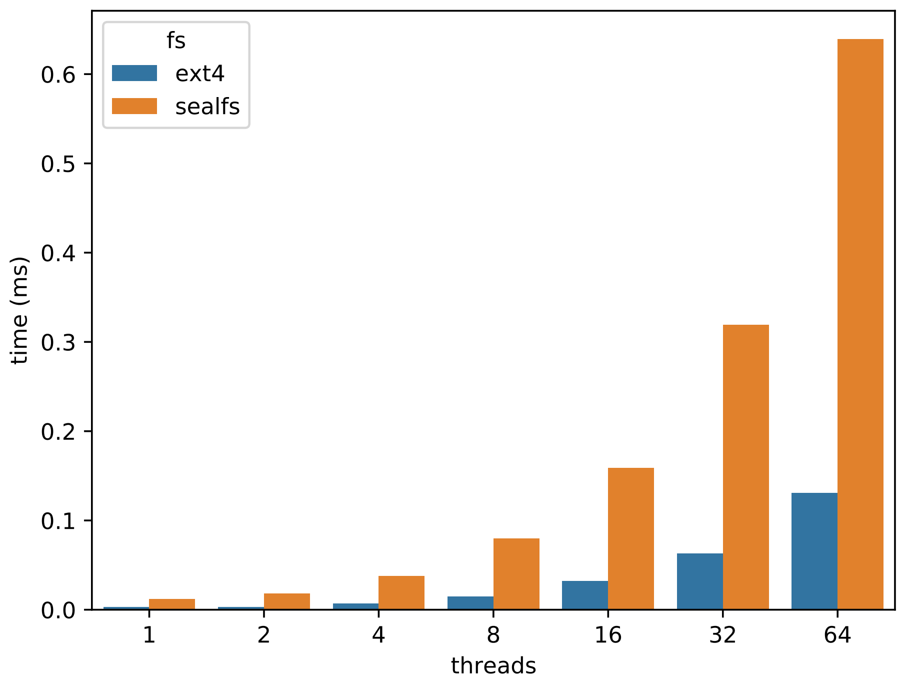

Figure 4: Filebench reported time for 100 byte write operations on SealFS, SSD disk and different numbers of concurrent threads with shared log files

Before executing the 1000 rounds, each process completes a cache heating phase of 500 write operations.

6.2.2 Results

We present the results of the experiment with box-and-whisker plots[10]. Time is always expressed in nanoseconds and data size in bytes.

Figure 7 shows the time to complete 100 byte write operations (approximately a line for a common plain text log) using the SSD disk. It compares the standard Ext4 file system (NOSEALFS) and SealFS stacked over it (SEALFS). Each process writes its private log file. Figure 8 shows the outliers for the same results.

Figure 9 presents the time only for SealFS with different write sizes, comparing the two disk types. The number of processes is fixed to 8. Each

[10]The plot shows the quartile of the dataset, the whiskers extend to show the rest of the distribution and the center line shows the median.

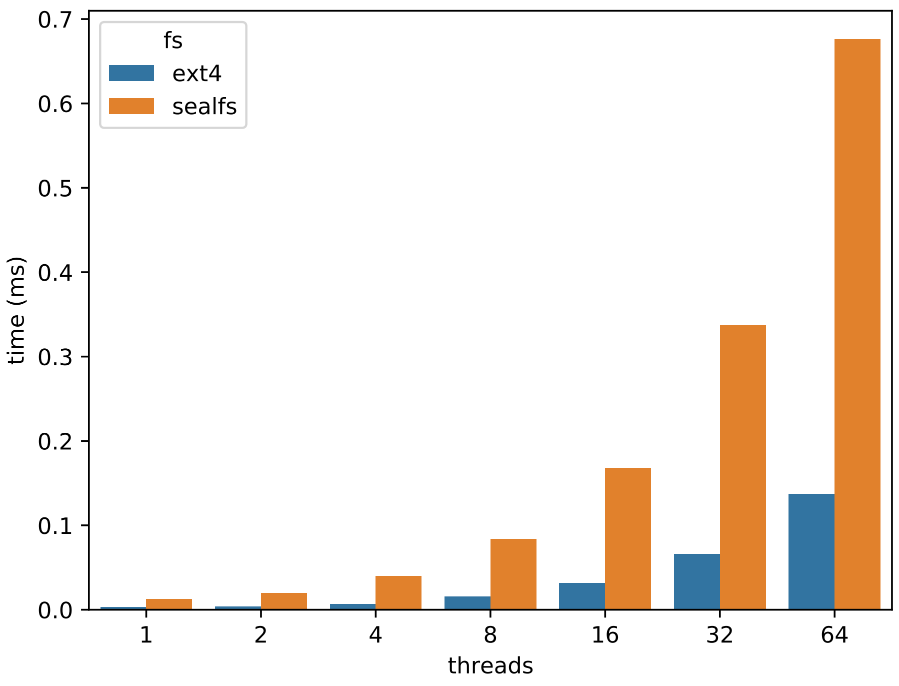

Figure 5: Filebench reported time for 100 byte write operations on SealFS, NOSSD disk and different numbers of concurrent threads with shared log files

process writes its private log file.

Figure 10 shows the time for different number of processes, comparing the results when using private (exclusive log file for each process) and shared log files.

Figure 11 compares synchronous and asynchronous I/O for $SEAL_{log}$ and K_{α}. It shows the times for only one process, two different write sizes and both disk types.

6.3 Analysis and Discussion

As shown by Figures 2, 3, 4, 5, 7 and 8, the overhead of using SealFS is notable but still manageable. In the most common scenario, SealFS is, at worst, one order of magnitude slower. The minibenchmark results show that the median for SealFS is 17205 ns; without SealFS, it is 1680 ns. The means are 17205 ns and 3385 ns respectively. These results are compatible with the

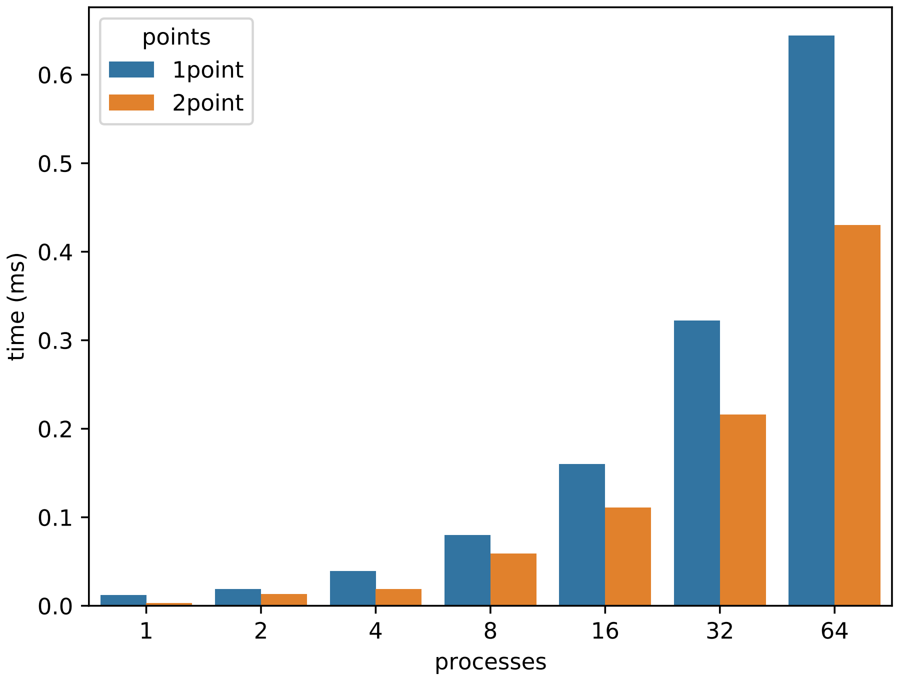

Figure 6: Filebench reported time for 100 byte write operations for concurrent processes, SSD disk, private log files and different number of SealFS instances (1 or 2 mount points).

results obtained with *filebench*, but let us analyze the behavior more closely.

The overhead we observe is mostly caused by the mutual exclusion to access $SEAL_{log}$ and K_α. When the applications share the log file, the synchronization to access the shared file also has a cost (we can see the dispersion in Figure 10). We have not made attempts to optimize this synchronization by making the critical region smaller, and it should not be difficult to make this better.

Figure 8 shows that when the contention is high (i.e. there are many concurrent processes trying to acquire the mutex), some write operations take a long time (up to 490 ms in the worst case, with 64 concurrent processes). If there is a context-switch while a process has the mutex acquired, all processes trying to acquire the mutex will block. The bigger the contention, the higher the probability of this happening. As shown in figure 6, this problem can be mitigated by mounting different instances of SealFS for different applications/processes.

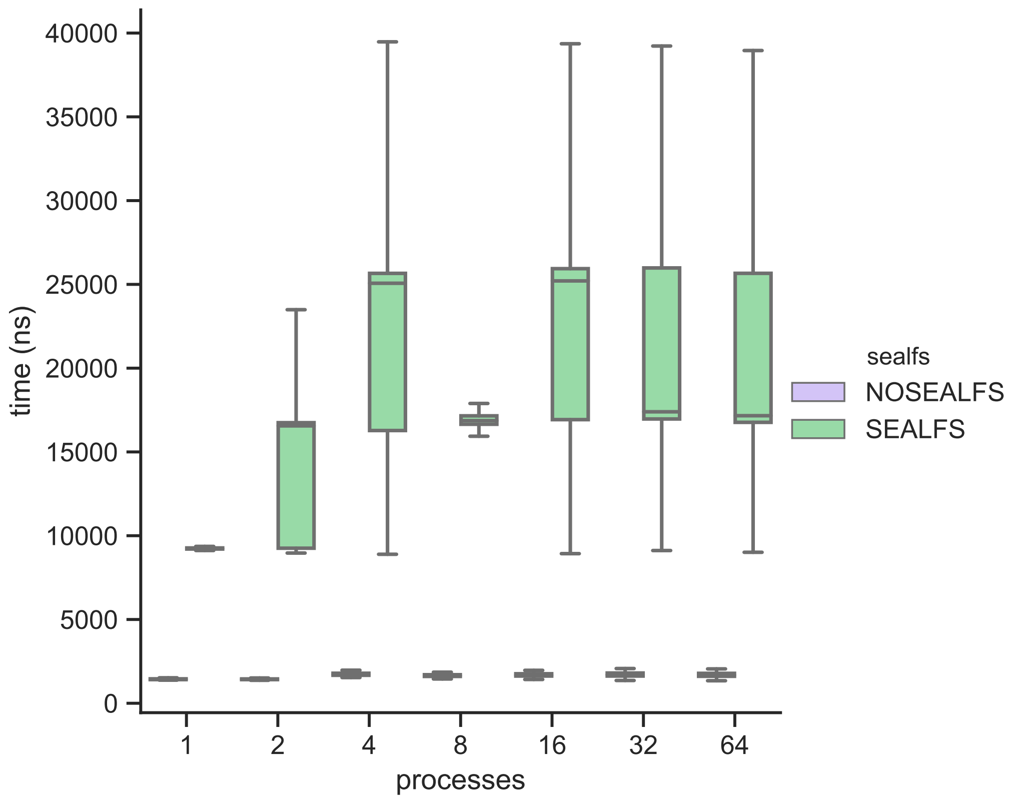

Figure 7: Time for 100 byte write operations, SSD disk and private log files. To the left and right of each category shown on the X axis are two different experimental results. To the left of the X ticks the whiskers depict NOSEALFS, and to the right SEALFS.

We can observe in Figure 9 that the type of disk is not critical for SealFS if I/O is asynchronous. Variance differs in each case, but the median is similar. This is the effect of the system's file cache and I/O optimization. Files are cached in RAM, so we don't pay the costs of writing the data synchronously to the disk. Moreover, files (append-only logs, $SEAL_{log}$ and K_α) are mostly accessed sequentially. Modern drives and operating systems are optimized for sequential access [54]. Writes to files opened as append-only are specially easy to optimize and Linux takes full advantage of this.

There are no clear differences between sharing the log file and using a private file (see Figures 2, 3, 4, 5 and 10). The reason is that the bottleneck is the mutual exclusion to access $SEAL_{log}$ and K_α, not the mutual access to write the shared file. In both cases (shared or private), the dispersion is very big for some executions. Note again that this is a consequence in part of the

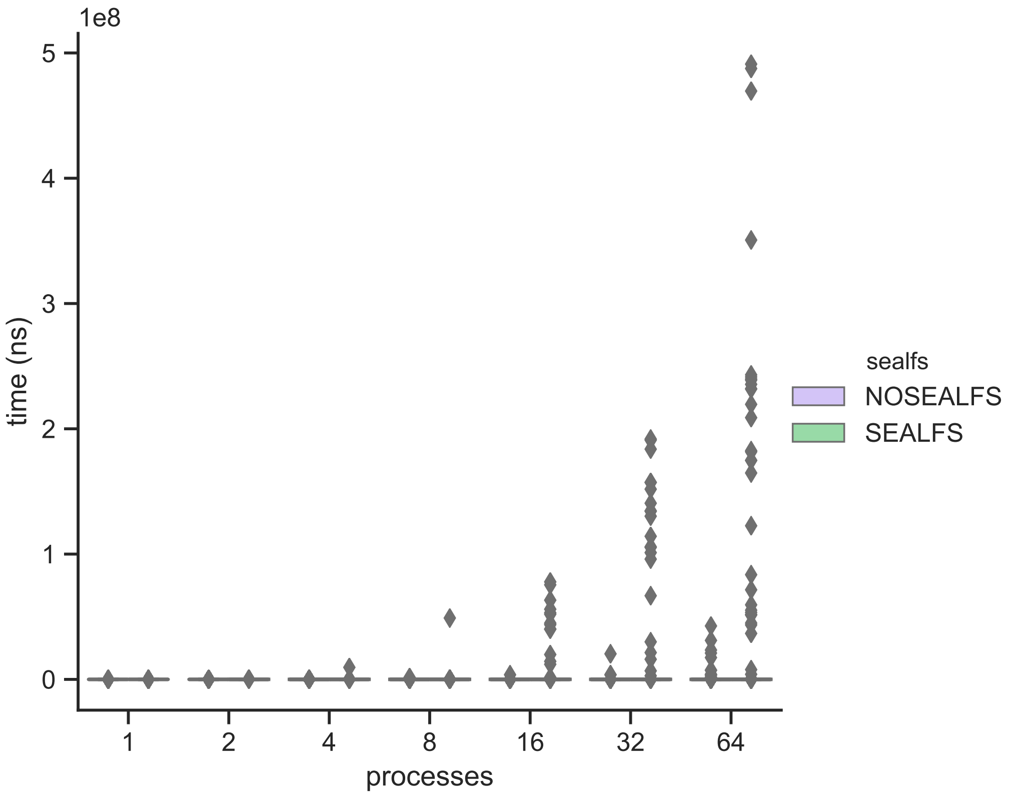

Figure 8: Results shown in figure 7 with the outliers. To the left and right of each category shown on the X axis are two different experimental results. To the left of the X ticks the whiskers depict NOSEALFS, and to the right SEALFS.

lack of optimization in the mutual access exclusion algorithm.

Forcing synchronous I/O for $SEAL_{log}$ and K_α is extremely expensive (see Figure 11), for both disk types. Note that this figure shows the result for only one process. For concurrent processes, it is not reasonable to wait for I/O completion while holding the mutex. The main disadvantage of using asynchronous I/O is that, in case of a system crash, the probability of getting a corrupted log is higher (the window is wider).

In general, using SealFS is expensive and it is not suitable to perform intensive I/O over the underlying files. Nevertheless, SealFS is intended for securing log files. Log files should not be under intensive I/O by design.

Spending approximately 0.02 ms for logging a 100 character text line seems reasonable for most applications that would require synchronous tamper-evident logs. Note that logging is a very small part of what an application

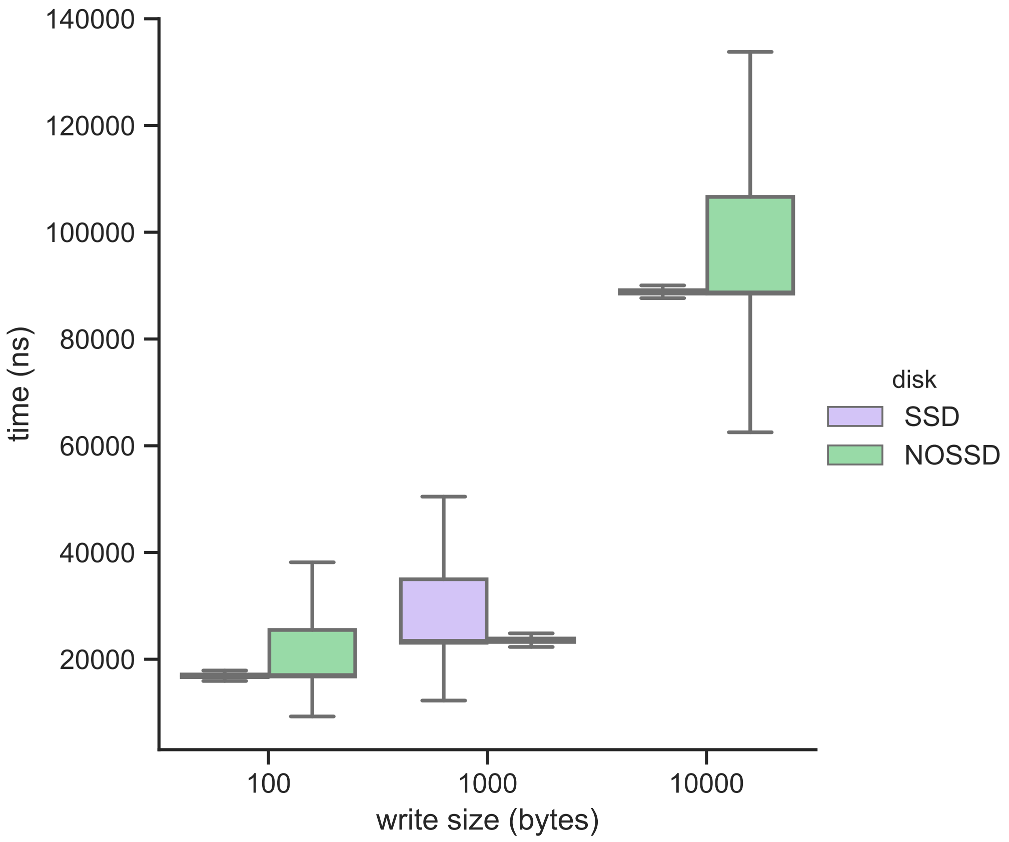

Figure 9: Time for SealFS with different write operation sizes, 8 processes with private log files. To the left and right of each category shown on the X axis are two different experimental results. To the left of the X ticks the whiskers depict SSD, and to the right NOSSD.

does. Normally, the accumulated time spent on logging is negligible. Nevertheless, application logging is normally asynchronous (either it is buffered or completely asynchronous on multi-threaded systems). Therefore, an approximately $10x$ overhead for write operations does not mean that deploying SealFS on an application would make its logging operations approximately $10x$ slower.

Verification is fast. For example, a single log file of 1.5 MB with 1500 records in $SEAL_{log}$ was verified in 0.011 s. A hundred 1.5 MB log files with 150000 records in $SEAL_{log}$ were verified in 0.989 s. The write size was 1000 bytes in both cases.

Verification will not happen often, only at audit. Nevertheless, time may be of the essence if we are trying to decide if the system has been compro-

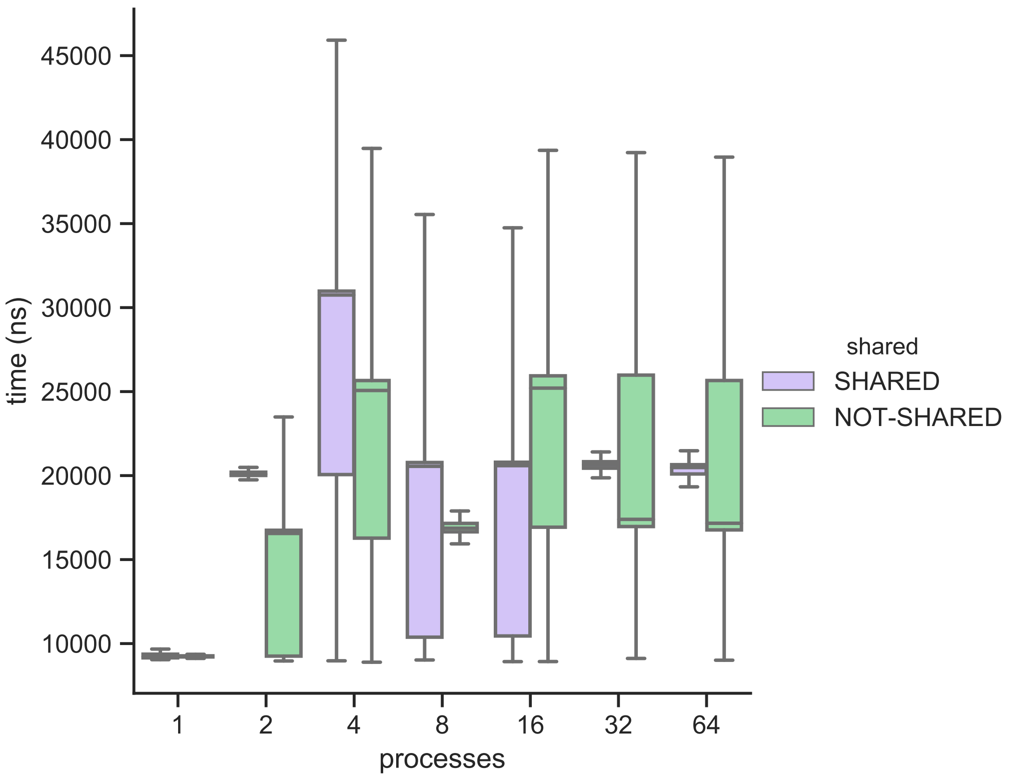

Figure 10: Time for 100 byte write operations on SealFS, SSD disk and different numbers of concurrent processes, comparing the results for shared and private files. To the left and right of each category shown on the X axis are two different experimental results. To the left of the X ticks the whiskers depict SHARED, and to the right NOT-SHARED.

mised. Making the verification fast can shorten the downtime. Several efforts in the literature have been spent trying to make verification in ratchets faster (see, for example, [7]).

As anecdotal data, for our main Linux server, which provides various network services with a public IP address and has an uptime of 10 days, the `journalctl` command reports 1036046 lines of logs. This means that, at this rate, if we used a 64 bytes key HMAC with a keystream of 32 GB our scheme could run for around 14 years without running out of secrets. As another example, publicly available Hadoop (HDFS) logs comprise approximately 4 million log entries per day [55]. In this case, a 32 GB keystream would last approximately 1.2 years. For some more money, $150 a 1 TB SSD hard disk will never run out of secrets even for applications that do almost nothing but

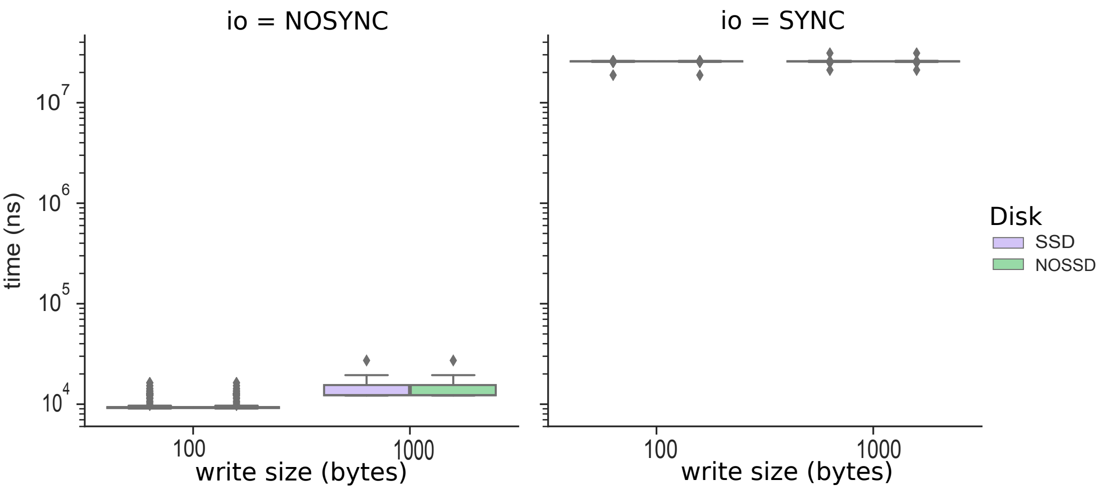

Figure 11: Time for synchronous and asynchronous I/O, only one process. To the left and right of each category shown on the X axis are two different experimental results. To the left of the X ticks the whiskers depict SSD, and to the right NOSSD. Note the logarithmic axis.

log continuously. If the trend continues, storage will only get cheaper.

7 Conclusions

In this paper we present a new scheme to provide tamper-evident logs in disconnected or loosely connected systems that does not depend on specialized hardware. This scheme is simple and takes advantage of the current prices of storage: our prototype can authenticate up to $1.6e9$ write operations with a \$5 32 GB external flash drive.

We also present a prototype implementation of the scheme, SealFS. It follows a novel approach to implement tamper-evident logs, with forward integrity based on a stackable file system. SealFS enables a transparent way to protect the logs of existing software without requiring any modification of applications, libraries or frameworks. As a proof of concept, we have a working version integrated in a conventional operating system like Linux, taking into account all the details necessary to make it work.

In addition, we provide some experimental results that show that the approach performs well on limited hardware. Two different benchmarks have been used to evaluate the prototype: a standard benchmark for file systems, *filebench*, and a custom minibenchmark. As expected, our approach is not suitable for intensive I/O applications, but it is appropriate to provide tamper-evident logs to critical applications.

Future work includes a new ratchet/keystream hybrid scheme and the evaluation of different strategies for better performance: optimizing the algorithm to reduce the critical section, evaluating other fine-grained synchronization schemes and testing new mechanisms to improve I/O performance (e.g. tailored caching mechanisms, etc.).

The source code of the SealFS prototype can be downloaded from:

`https://gitlab.etsit.urjc.es/esoriano/sealfs/tree/master`

References

[1] L. Zeng, Y. Xiao, H. Chen, B. Sun, and W. Han, "Computer operating system logging and security issues: a survey," *Security and Communication Networks*, vol. 9, no. 17, pp. 4804–4821, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1677

[2] M. Bellare and B. S. Yee, "Forward integrity for secure audit logs," University of California at San Diego, Tech. Rep., 1997.

[3] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs, "Ratcheted encryption and key exchange: The security of messaging," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 619–650.

[4] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267553

[5] J. Kelsey and B. Schneier, "Minimizing bandwidth for remote access to cryptographically protected audit logs," in *Recent Advances in Intrusion Detection*, 1999, pp. 9–9.

[6] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 2, p. 159–176, May 1999. [Online]. Available: https://doi.org/10.1145/317087.317089

[7] A. Sinha, L. Jia, P. England, and J. R. Lorch, "Continuous tamper-proof logging using tpm 2.0," in *Trust and Trustworthy Computing*, T. Holz and S. Ioannidis, Eds. Cham: Springer International Publishing, 2014, pp. 19–36.

[8] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Trans. Storage*, vol. 5, no. 1, Mar. 2009. [Online]. Available: https://doi.org/10.1145/1502777.1502779

[9] A. Yavuz, P. Ning, and M. Reiter, "Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging," in *Financial Cryptography*, 2012.

[10] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," *Network and Distributed System Security Symposium*, Jan 2020. [Online]. Available: http://par.nsf.gov/biblio/10146530

[11] H. Nguyen, B. Acharya, R. Ivanov, A. Haeberlen, L. T. X. Phan, O. Sokolsky, J. Walker, J. Weimer, W. Hanson, and I. Lee, "Cloud-based secure logger for medical devices," in *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, 2016, pp. 89–94.

[12] V. Karande, E. Bauman, Z. Lin, and L. Khan, "Sgx-log: Securing system logs with sgx," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 19–30. [Online]. Available: https://doi.org/10.1145/3052973.3053034

[13] Dogtag, "Managing Signed Audit Logs," https://www.dogtagpki.org/wiki/Signed_Audit_Log, 2019, [Online; accessed may-2019].

[14] L. Zeng, H. Chen, and Y. Xiao, "Accountable administration and implementation in operating systems," in *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, Dec 2011, pp. 1–5.

[15] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok, "I3fs: An in-kernel integrity checker and intrusion detection file system," in *Proceedings of the 18th USENIX Conference on System Administration*, ser. LISA '04. USA: USENIX Association, 2004, p. 67–78.

[16] B. Chou, K. Tatara, T. Sakuraba, Y. Hori, and K. Sakurai, "A secure virtualized logging scheme for digital forensics in comparison with kernel module approach," in *2008 International Conference on Information Security and Assurance (isa 2008)*, April 2008, pp. 421–426.

[17] Loggly, "Loggly: Remote Logging Service," https://www.loggly.com/solution/remote-logging-service/, 2019, [Online; accessed may-2019].

[18] Stackdriver, "Stackdriver Logging," https://cloud.google.com/logging/, 2019, [Online; accessed may-2019].

[19] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 317–334.

[20] T. Pulls and R. Peeters, "Balloon: A forward-secure append-only persistent authenticated data structure," *IACR Cryptology ePrint Archive*, vol. 2015, p. 7, 2015. [Online]. Available: https://eprint.iacr.org/2015/007

[21] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger, "Self-securing storage: Protecting data in compromised system," in *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation - Volume 4*. USA: USENIX Association, 2000.

[22] M. Rosa, J. P. Barraca, and N. P. Rocha, "Logging integrity with blockchain structures," in *New Knowledge in Information Systems and Technologies*, Á. Rocha, H. Adeli, L. P. Reis, and S. Costanzo, Eds. Cham: Springer International Publishing, 2019, pp. 83–93.

[23] H. Wang, D. Yang, N. Duan, Y. Guo, and L. Zhang, "Medusa: Blockchain powered log storage system," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, 11 2018, pp. 518–521.

[24] LogSentinel, ","https://logsentinel.com/, 2019, [Online; accessed may-2019].

[25] Guardtime, "Blockchain Backed Log Assurance," https://guardtime.com/solutions/blockchain-backed-log-assurance, 2019, [Online; accessed may-2019].

[26] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016, pp. 164–178.

[27] J. Holt and K. Seamons, "Logcrypt: Forward security and public verification for secure audit logs," in *IACR Cryptol. ePrint Arch.*, 2005.

[28] A. A. Yavuz and P. Ning, "Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 219–228.

[29] G. Hartung, B. Kaidel, A. Koch, J. Koch, and D. Hartmann, "Practical and robust secure logging from fault-tolerant sequential aggregate signatures," in *ProvSec*, 2017.

[30] G. Hartung, "Attacks on secure logging schemes," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 95, 2017.

[31] S. Han, W. Shin, J.-H. Park, and H. Kim, "A bad dream: Subverting trusted platform module while you are sleeping," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1229–1246.

[32] A. Czeskis and J. Lang, "Fido nfc protocol specification v1.0," FIDO Alliance Proposed Standard, 2015.

[33] J. Ehrensvärd and J. Kemp, "Fido hid protocol specification v1.0," FIDO Alliance Proposed Standard, 2015.

[34] A. Czeskis and J. Lang, "Fido bluetooth protocol specification v1.0," FIDO Alliance Proposed Standard, 2015.

[35] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," IETF, RFC 2104, Feb. 1997. [Online]. Available: http://tools.ietf.org/rfc/rfc2104.txt

[36] R. C. Merkle, "Protocols for public key cryptosystems," in *1980 IEEE Symposium on Security and Privacy.* IEEE, 1980, pp. 122–122.

[37] D. S. Rosenthal, D. Rosenthal, E. L. Miller, I. Adams, M. W. Storer, and E. Zadok, "The economics of long-term digital storage," in *The Memory of the World in the Digital Age: Digitization and Preservation*, Sep. 2012.

[38] G. S. Vernam, "Cipher printing telegraph systems: For secret wire and radio telegraphic communications," *Journal of the AIEE*, vol. 45, no. 2, pp. 109–115, 1926.

[39] C. Braz and J.-M. Robert, “Security and usability: the case of the user authentication methods,” in *Proceedings of the 18th Conference on l’Interaction Homme-Machine*, 2006, pp. 199–203.

[40] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano, “The quest to replace passwords: A framework for comparative evaluation of web authentication schemes,” in *2012 IEEE Symposium on Security and Privacy.* IEEE, 2012, pp. 553–567.

[41] ——, “Passwords and the evolution of imperfect authentication,” *Communications of the ACM*, vol. 58, no. 7, pp. 78–87, 2015.

[42] D. Wang and P. Wang, “Two birds with one stone: Two-factor authentication with security beyond conventional bound,” *IEEE transactions on dependable and secure computing*, vol. 15, no. 4, pp. 708–722, 2016.

[43] D. DeFigueiredo, “The case for mobile two-factor authentication,” *IEEE Security & Privacy*, vol. 9, no. 5, pp. 81–85, 2011.

[44] E. Zadok and I. Badulescu, “A stackable file system interface for linux,” in *In LinuxExpo Conference Proceedings*, 1999, pp. 141–151.

[45] E. Barker, “Recommendation for key management,” NIST, NIST Technical Report 56, May 2020. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-57pt1r5

[46] P. Gutmann, “Secure deletion of data from magnetic and solid-state memory,” in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM’96. USA: USENIX Association, 1996, p. 8.

[47] “U.S. National industrial security program operating manual DoD 5220.22-M.” United States Department of Defense National Industrial Security Program, 2006.

[48] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, “Reliably erasing data from flash-based solid state drives,” in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1960475.1960483

[49] W. Bhat and S. Quadri, “Restfs: Secure data deletion using reliable & efficient stackable file system,” in *2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (SAMI).* IEEE, 2012, pp. 457–462.

[50] N. Joukov and E. Zadok, "Adding secure deletion to your favorite file system," in *Third IEEE International Security in Storage Workshop (SISW'05).* IEEE, 2005, pp. 8–pp.

[51] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *USENIX; login*, vol. 41, no. 1, pp. 6–12, 2016.

[52] "Pitfalls of TSC usage," http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/.

[53] G. Paoloni, *White paper: How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*, Intel, 2010. [Online]. Available: http://www.intel.es/content/www/es/es/embedded/training/ia-32-ia-64-benchmark-code-execution-paper.html

[54] M. Hinner, "Filesystems HOWTO: Extended filesystems (Ext, Ext2, Ext3)," https://www.tldp.org/HOWTO/Filesystems-HOWTO-6.html , 2007, [Online; accessed may-2019].

[55] M. Landauer, F. Skopik, M. Wurzenberger, and A. Rauber, "System log clustering approaches for cyber security applications: A survey," *Computers and Security*, vol. 92, p. 101739, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820300250