International Journal of Information Security (2023) 22:447–466
https://doi.org/10.1007/s10207-022-00643-1

REGULAR CONTRIBUTION

SealFSv2: combining storage-based and ratcheting for tamper-evident logging

Gorka Guardiola-Múzquiz[1] · **Enrique Soriano-Salvador**[1]

Published online: 6 December 2022

Abstract
Tamper-evident logging is paramount for forensic audits and accountability subsystems. It is based on a forward integrity model: upon intrusion, the attacker is not able to counterfeit the logging data generated before controlling the system. There are local and distributed solutions to this problem. Distributed solutions are suitable for common scenarios, albeit not appropriate for autonomous and loosely connected systems. Moreover, they can be complex and introduce new security issues. Traditional local tamper-evident logging systems use cryptographic ratchets. In previous works, we presented SealFS (from now on, SealFSv1), a system that follows a radically different approach for local tamper-evident logging based on keystream storage. In this paper, we present a new version, SealFSv2, which combines ratcheting and storage-based log anti-tamper protection. This new approach is flexible and enables the user to decide between complete theoretical security (like in SealFSv1) and partial linear degradation (like in a classical ratchet scheme), exchanging storage for computation with user-defined parameters to balance security and resource usage. We also describe an implementation of this scheme. This implementation, which showcases our approach, is an optimized evolution of the original `sealfs` Linux kernel module. It implements a stackable file system that enables transparent tamper-evident logging to all user space applications and provides instant deployability. Last, we present a complete performance evaluation of our current implementation and a fair performance comparison of the two opposite approaches for local tamper-evident logging (i.e., storage-based vs. ratcheting). This comparison suggests that, on current systems and general-purpose hardware, the storage-based approach and hybrid schemes perform better than the traditional ratchet approach.

Keywords Cybersecurity · Logging · File system · Tamper-evident · Verification · Authentication · Forensics

1 Introduction

A tamper-evident logging system protects the integrity of logging data generated in the past, following the forward integrity model [1]. Upon intrusion, attackers may be able to change the logging data generated in the past, but the tamper-evident system will detect any integrity violation. This way, forensic auditors are capable of detecting and discarding counterfeit logging data.

There are different solutions to provide logging data integrity that are based on distributed systems, such as traditional logging servers or modern systems based on distributed ledger technology (like blockchain). Nevertheless, distributed solutions are not suitable or desirable for loosely connected or disconnected systems. For example, autonomous robotic systems that operate at isolated areas must use a local tamper-evident system for their accountability subsystem. An accountability subsystem manages the robot information (sensor/actuator data, cognitive information, etc.) to explain its behavior and determine whether it has been caused an accident or malfunction: Any accountability data is useless if it can be faked by an attacker. Our work focuses on local solutions.

Traditional local tamper-evident systems are based on *cryptographic ratchets* [2,3] (i.e., hash chains, PRF chains, linear Merkle trees, etc.). This approach is rather old [1,4–6]. Nevertheless, several recent works have evolved this scheme to take advantage of new security hardware, etc.

✉ Gorka Guardiola-Múzquiz
gorka.guardiola@urjc.es

Enrique Soriano-Salvador
enrique.soriano@urjc.es

[1] Universidad Rey Juan Carlos, Madrid, Spain

In contrast, we designed and implemented a system that follows an opposite approach [7]. SealFS (from now on, SealFSv1) is a Linux kernel module that implements a stackable file system that enables tamper-evident logging. It uses a long pre-generated keystream stored on cheap, large disks. This long keystream provides keys to authenticate the data appended to log files. Later, in audit time, a copy of the keystream is used to verify the logging data. The main advantage of this approach is its simplicity.

These two approaches are complete opposites. SealFSv1 is closer to the extreme of the spectrum similar to the one-time pad (OTP) encryption: the keys for authenticating the logging data are extracted from a pre-generated cryptographically-secure pseudorandom stream stored on a high capacity device. The pure ratchet approach is close to the other extreme: continuous key derivation. The former may waste a lot of storage space and be I/O bound. The latter causes linearly degrading security [1,7] and may be CPU bound.

There is an open question regarding performance (i.e., the time needed to append data to log files and the time needed to verify the log files): *Which approach performs better on current general-purpose hardware?* As far as we know, this question remains unanswered.

In our previous work, we provided a quantitative performance analysis of SealFSv1 [7], but we were not able to compare it with ratchets in a controlled way. The SealFSv1 implementation (a Linux kernel module that implements a file system) and the different ratchet implementations were too different for conducting fair comparative experiments in order to answer the previous question.

In this paper, we present SealFSv2, a new version that combines both approaches. SealFSv2 uses a pre-generated keystream stored in a device, but it is also able to derive a set of keys by ratcheting. How many keys are obtained per piece of the keystream is configurable by the user. More keys derived means that the keystream will last longer (or occupy less) but will provide less (theoretical) security[1]. All the other characteristics (dependencies, assumptions) stay the same as in SealFSv1.

Given that SealFSv2 can be configured to behave as SealFSv1, a ratchet and a hybrid, it can be used to conduct comparative experiments to try to answer the previous question.

1.1 Contributions

The contributions of this paper are:

- A novel ratchet/storage-based hybrid scheme for tamper-evident logs. As far as we know, no other system follows a similar strategy combining both approaches to enable forward integrity.
- The description of a performance-tuned open-source implementation of this scheme: a Linux kernel module that provides a stackable file system and userspace tools for verification. The description includes details about fine-grained concurrency aspects and optimizations.
- The evaluation of our prototype, providing performance data acquired from a custom microbenchmark and a standard benchmark (*Filebench* [8]).
- A fair and solid comparison of the two opposite approaches (ratchet vs. storage-based). Given that our system implements both approaches and it's configurable, we are able to compare them in a controlled environment.

1.2 Organization

The paper is organized as follows: Sect. 2 discusses the state of the art, Sect. 3 presents our approach, Sect. 4 describes the implementation of SealFSv2, Sect. 5 shows its evaluation and the comparative experiments, and Sect. 6 presents conclusions and future work.

2 Related work

There is a vast body of related work. We described the main differences of SealFSv1 and closed related work elsewhere [7]. This is a succint description of the different solutions presented in the literature for logging integrity:

- **Non-cryptographic models** for accounting (e.g., the system presented by Zeng et al. [9]) and **integrity checkers** (e.g., I^{3}FS [10]). These systems are based on policies and depend on a trusted platform (i.e., kernel, hypervisor). Thus, they do not comply with a forward integrity model if the whole system is compromised. Other solutions based on virtualization (for example, see the scheme proposed by Chou et al. [11]) have the same problem.
- **Distributed solutions** have a fundamental issue: they do not work for loosely connected or disconnected systems. There are multiple distributed solutions: traditional logging servers, cloud-based logging services (see for example [12,13]), self-securing storage like S4 [14], server-based systems that use history trees and randomized binary search trees (hash treaps) [15,16] and systems based on distributed ledger technology (DLT), or Blockchain [17–21]. As stated before, we focus on local tamper-evident logs.
- **Traditional local solutions** are based on **cryptographic ratchets** [1–6,22]. A ratchet is a one-way derivation func-

[1] For further details and discussion, see Sect. 4.4 of the original SealFS paper [7].

tion, i.e., a chain of non-reversible pseudorandom functions, secure hashes, authentication codes (e.g., HMAC), etc. A ratchet starts with a secret (or several secrets) and provides a sequence of secrets, overwriting the used ones, that will be used to authenticate data written to log files. Given that the chain function is not reversible, the attacker cannot find a secret used before she takes control of the system. Therefore, after the intrusion, the attacker is not able to silently fake already authenticated logging data. The ratchet approach was proposed three decades ago. Since then, some authors have evolved it. For example, Ma et al. [23] proposed a scheme that combines authentication tags to be more efficient in terms of storage. There are other works in the same line (e.g., [24–26]). Some of these systems have security issues [27]. SealFSv1 used a pure storage-based approach [7], storing all the keys used to authenticate the logs. This approach was sketched by Bellare et al. *Random Key FI-MAC* scheme [1], but it was not feasible in 1997. As far as the authors know, it was never implemented. Currently, changes in storage technology have made it practical to implement. Note that SealFSv1's approach is not the same (it never reuses the keys, its epoch is just one log message) and the implementation is also novel. SealFSv2 combines the ratchet approach with SealFSv1's storage-based approach. A recent tamper-evident logging system based on the ratchet approach is KennyLoggings [28]. In this work, Paccagnella et al. describe a race condition present in most tamper-evident logging systems. KennyLoggings provides forward integrity to Linux Audit (`auditd`), a Linux subsystem used to trace kernel events[2]. KennyLoggings is implemented inside the kernel and avoids the cited race condition, satisfying the *synchronous integrity* property. SealFSv2 runs in the kernel, but it does not authenticate kernel events. It is a general purpose tool. The goal is to provide forward integrity for application logging data. SealFSv2 also satisfies the *synchronous integrity* property (i.e., it is not vulnerable to the race condition). Moreover, it does not depend on other kernel mechanisms or subsystems like Linux Audit. Note that the overhead caused by Linux Audit is very high [29] and it affects the whole system. On the other hand, the mechanisms of SealFSv2 only affect processes appending data to a few selected files (the protected logs), the rest of the system remain unaffected.

– Systems based on **specialized devices and secure hardware**. Old systems used WORM (Write Once Read Many) devices and printers to enable logging integrity. Modern systems take advantage of secure hardware. For example, Sinha et al. [30] evolved the ratchet approach to address some of the issues of previous proposals and use TPM 2 (Trusted Platform Module). Other works use secure enclaves with sealing/unsealing primitives to provide integrity and confidentiality of logging data [31–33]. SealFSv2 does not depend on specialized hardware to enable tamper-evident logging.

In addition, some systems are extremely complex, while SealFSv2 is focused on simplicity. As Schneier states, *"complexity is the worst enemy of security, and our systems are getting more complex all the time"* [34].

For example, Black Block Recorder (BBR) [17] is an event data recorder for robotics. It follows a very complex approach based on secure hardware, digital signatures, public key infrastructure (PKI), ratchets, smart contracts and distributed ledgers. The robot executes a recorder process on a secure enclave that captures the events, derives a linked integrity proof based on a ratchet (i.e., HMAC chain), signs and sends the data to a storage subsystem that binds the data to the robot's public identity with a digital certificate. Then, the data are sent to another subsystem, the *validator*. The *validator* is based on the Hyperledger Sawtooth [35], a complex open-source ecosystem of blockchain development that provides distributed ledgers and supports several consensus algorithms (including a byzantine fault-tolerant one), parallelizable transactions, and so on. This *validator* keeps the distributed ledger state and cooperates with other robots' *validators*.

This work is very ambitious and its goals are different from ours. SealFSv2 aims to provide a simple and understandable local system to enable general purpose tamper-evident logging (suitable for robotics or any other applications) without depending on complex architectures (e.g., Hyperledger), specialized hardware, secure coprocessors, etc.

3 Scheme

3.1 Main ideas

The ideas behind SealFSv1 and SealFSv2 are similar. Both use two different storage devices: α and β. α is an storage device available when the system is running. β is an external storage device that will be used to verify the logs in the future (only available at audit time).

A pseudo-random keystream K is securely generated and a copy is stored in α and β: K_α is the keystream stored in α and K_β is the keystream stored in β. Both have a header with some metadata:

– $Kheader.id$: an identification number for the keystream.
– $Kheader.off$: the current position, or *offset*, of the keystream. It is initialized to zero.

[2] Note that Linux Audit is not used for application data logging and does not replace user space logging frameworks, `syslog`, etc.

After configuring the system, β is physically disconnected and kept in a safe place.

When the system starts to execute (i.e., normal operation), a file ($SEAL_{log}$) is used to store the authentication data and metadata for the data appended to the log files. Chunks of K_α are read and used to authenticate the data written in log files. Used chunks are removed from α (they are *burnt*). Thus, K_α will be burnt from offset 0 to offset $K_\alpha header.off - 1$,

SealFSv2 uses an extra parameter, $NRATCHET$. It is not stored in the header, it must be provided as an argument when the system starts to execute. This value is the number of derived keys per chunk read from K_α. When $NRATCHET$ is greater than 1, we say that *the key is ratcheted*.

In audit time, the volumes with the log files, both α and β, are attached to a trusted computer. Then, SealFSv2 is able to detect if some parts of the log files that were generated in execution time (before the possible exploitation) have been modified.

Note that, upon full system compromise, the leaked keys are the same for any $NRATCHET$ value. None of the keys used before the attacker elevates privileges are leaked, because they have been burnt or ratcheted. All the keys used since the attacker elevates privileges will be compromised: The adversary could get them directly from the keystream (disk) or from the internal state of the ratchet (memory).

3.2 Threat model

The threat model is similar to that in SealFSv1 [7]. Briefly:

- The asset is the logging data generated while the system is not compromised. We focus on the *forward integrity model* [1].
- Before the exploitation, the adversary has no access to K (α or β).
- Upon exploitation, the adversary has total control of the system (software and hardware).

Storage devices (α and β) must be big enough to store a long keystream. There is also an important assumption: The deletion procedure used by the system to burn K_α is secure. Once data is erased, it cannot be recovered by the attacker.

For SealFSv2, there is a new assumption: The adversary may be able to compromise the security of keys derived by the ratcheting algorithm due to linear degradation [1]. The threat grows proportionally with the value of $NRATCHET$.

3.3 Appending data to log files

The system needs extra space to store five integer values and a secure digest per write operation (independently of the number of bytes written).

$HMAC(key, msg)$ is a secure message authentication code algorithm based on a keyed-hash [36]. 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 file). 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 (see algorithm 1):

1. The data is appended to the log file.
2. The current offset of K_α is read from its header.
3. If the ratchet is *consumed* (i.e., this iteration is a multiple of $NRATCHET$) a chunk of keystream (C_i) is read from K_α. We use a circular counter initialized to 0, $Roff$, to detect if the ratchet is consumed (when $Roff$ equals 0, it is consumed). 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. The updated offset of K_α is written to its header. Then, the corresponding chunk of K_α must be *burnt*. This is done asynchronously by another *burner* process when notified. This process is woken up whenever a piece of key must to be burnt and executes Algorithm 2.
4. If *the key is ratcheted*, the ratchet advances:

$$K = HMAC(K, Roff||NRATCHET)$$

   We will refer to this, which generates a new key computationally from the last one, as the *ratcheting algorithm* (see Algorithm 3 and Fig. 1).
5. 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 ($Coff_i$), $Roff$ and the data (D_i) is calculated ($HMAC_i$). K is used as the key. Later, the key K will be overwritten by the next key (either by ratcheting or reading from the file).
6. A record R with fields

$$[L, Loff_i, Dsz_i, Coff_i, Roff, HMAC_i]$$

   is created. Note that when the key is ratcheted, all records are implicitly labeled with the number of pending derived keys for the current chunk of keystream (C_i).
7. The record R is appended to $SEAL_{log}$.
8. The circular counter that represents the number of keys generated by this ratchet is incremented[3]:

$$Roff = (Roff + 1) \bmod NRATCHET$$

[3] Note that mod represents the modulo (remainder of the division operator).

Algorithm 1 describes this procedure. Note that the records of $SEAL_{log}$ are always sorted according to the key used (ratcheted or not). Being R_i and R_j two contiguous records in $SEAL_{log}$, K_i and K_j their corresponding keys, then K_j is always the next key for K_i: if the key is not ratcheted, K_i is the previous chunk of keystream; if the key is ratcheted, K_j is the first derivation from K_i (or the next chunk when the ratchet was consumed).

Algorithm 1 Write algorithm

1: append D_i to L at offset $Loff_i$
2: **if** $Roff == 0$ **then**
3: $Coff_i \leftarrow K_\alpha header.off$
4: $C_i \leftarrow K_\alpha[Coff_i \ldots Coff_i + Csz - 1]$
5: $K \leftarrow C_i$
6: $K_\alpha header.off \leftarrow Coff_i + Csz$
7: notify burner process to burnt (C_i) from K_α
8: **end if**
9: **if** $NRATCHET \neq 1$ **then**
10: $K \leftarrow HMAC(K, Roff||NRATCHET)$
11: **end if**
12: $H_i \leftarrow HMAC(C_i, L||Loff_i||Dsz_i||Coff_i||Roff||D_i)$
13: $R \leftarrow (L, Loff_i, Dsz_i, Coff_i, Roff, H_i)$
14: append R to $SEAL_{log}$
15: $Roff \leftarrow (Roff + 1)$ mod $NRATCHET$

Algorithm 2 Burning algorithm

1: $K_\alpha[Coff_i \ldots Coff_i + Csz - 1] \leftarrow RANDOM()$

Algorithm 3 Ratcheting algorithm

1: $K \leftarrow K_\alpha[Coff_i \ldots Coff_i + Csz - 1]$
2: **for** $i = Roff_{prev}, i < Roff$ **do**
3: $K \leftarrow HMAC(K, i||NRATCHET)$
4: **end for**

3.3.1 Concurrency

Note that write operations are concurrent (for the same log file and for different log files). The write algorithm must be executed with synchronization between different processes to preserve the integrity of $K_\alpha header.off$, the order of $SEAL_{log}$ and the relationship between $Roff$ and the key. This may be problematic.

In Algorithm 1, there are two atomic blocks: red block (line 1) and the blue block (lines 2-15). There could be *overruns* of concurrent processes performing write operations for the same log file L: a process A can overrun a process B between these two atomic blocks. The time window is short, but it is possible[4]. In other words, for the same file L, being $Loff_i < Loff_j$, a record R_i with offset $Loff_i$ could be written to $SEAL_{log}$ *after* a record R_j with offset $Loff_j$.

SealFSv2 takes care of that disorder, which is handled by the verification algorithm (see Algorithm 4). The write algorithm of the previous version [7] ignored this detail,

which is important in practice and affects the performance of the final implementation.

In practice, an overrun would happen before or after an attack: It is extremely improbable to happen in the meantime, cluttering a genuine write and a malicious write. Anyway, from the point of view of the auditor, both (genuine and malicious) should be considered post-exploitation records.

3.4 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 4 which also uses algorithms 5, 6, 7 and 8.

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.
- Records are ordered in $SEAL_{log}$ with respect to their key index (if the key comes from ratcheting, by $Roff$). That is, given two contiguous records, their corresponding keys must be *contiguous* (with respect to the combined index of their keys). In the verification algorithm, O_K and $Roff$ are used to check that. O_K is the current offset of the keystream. $Roff$ is the current offset of the ratchet.
- As explained before, records in $SEAL_{log}$ belonging to a log L are partially ordered (by $R.Loff$). To give the implementation wiggle space for performance, the log L can be slightly out of order with respect to $SEAL_{log}$. In any event, the data areas defined by all the records must be contiguous: if there is a gap, the verification always fails. An array, $O[]$, contains a position and a small heap[5] for each log file. This array is used to order the file efficiently while it is being read and check this invariant.

The corresponding chunk of the keystream is read from K_β (at offset $R.Coff$). If the key is not ratcheted (i.e., $NRATCHET = 1$), the key K is just the chunk. Else, the key is ratcheted to $R.Roff$ generating K. In this case, when

[4] As we will see later, in our implementation, that could only happen for concurrent write system calls from different processes, for the same file.

[5] A data structure used to implement efficient priority queues.

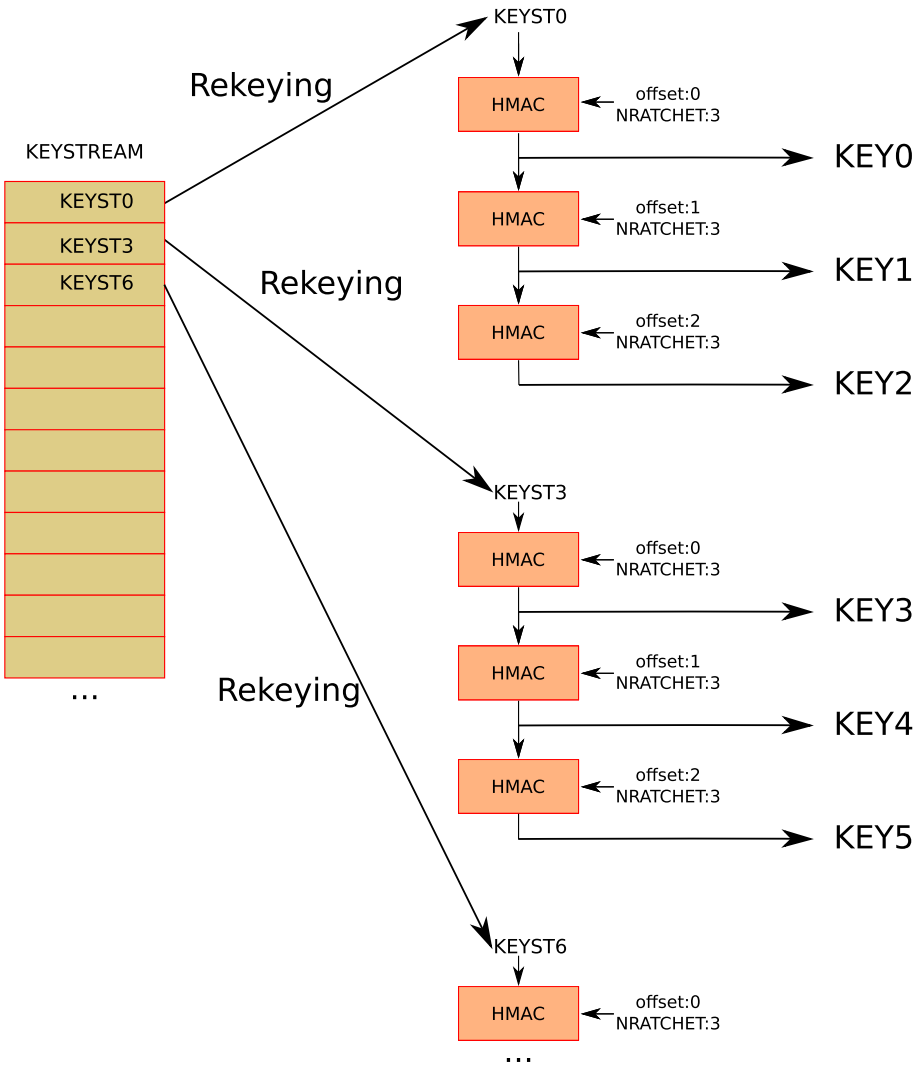


Fig. 1 Ratchet example with $NRATCHET = 3$

verifying many consecutive records, the last key is kept in a cache (see Algorithm 7). The key cache takes care of reading the keystream on demand and ratcheting it to the point as needed in an efficient manner.

The data described by a record R is read from the log file (file L, from position $R.Loff$, length $R.Dsz$) and the HMAC is regenerated using K. The new HMAC and the HMAC stored in R are compared. If they are not equal, the verification fails for L. Given that it is a secure HMAC, the attacker will not able to forge any HMAC of $SEAL_{log}$ or deduce the key K (ratcheted or not) for any record stored in $SEAL_{log}$.

The verification algorithm also fails if:

- The adversary removes any record for a log L from $SEAL_{log}$. All possible keys (ratcheted or not) already burnt in K_α must be used to verify records (the keys are extracted from K_β). Detection is easy: If a record is removed, a key will be unused.
- $SEAL_{log}$ is truncated and the corresponding lines of the log files are deleted. As in the previous point, this will be detected with the keys extracted from K_β.
- Any log file (L) is modified, truncated or shortened, or data are taken from it. In this case, the HMACs will not match.
- Any field of the records of $SEAL_{log}$ is modified. Again, the HMACs will not match.
- Two different records of $SEAL_{log}$ overlap for a file log L: If there is intersection between the areas defined by the records for L (defined by fields $Loff_i$ and Dsz_i), the verification will fail.

We use a heap for each log file L to check that there are no holes. In other words, we verify that the log file has been completely and contiguously covered by the records.

In the verification algorithm, contiguous records are popped from the corresponding heap in each iteration (see Algorithm 5). At the end of the verification algorithm, all

Algorithm 4 Verification algorithm

```
1: if size(K_α) ≠ size(K_β) or K_α header.id ≠ K_β header.id then
2:    FAIL()
3: end if
4: P ← K_α header.off
5: if K_α[P − Csz ... P − 1] = K_β[P − Csz ... P − 1] or K_α[P ... P + Csz − 1] ≠
   K_β[P ... P + Csz − 1] then
6:    FAIL()
7: end if
8: O_K ← 0
9: Roff ← 0
10: NRATCHET ← 1
11: for each record R of SEAL_log do
12:    if ((O_K/Csz)/NRATCHET) ∗ Csz ≠ R.Coff or Roff ≠ R.Roff then
13:       FAIL()
14:    end if
15:    if O[R.L] is not initialized yet then
16:       O[R.L] ← R.Loff
17:    end if
18:    if O[R.L].heap is full then
19:       FAIL()
20:    end if
21:    push R onto O[R.L].heap
22:    pop contiguous from O[R.L].heap
23:    if NRATCHET not detected then
24:       NRATCHET ← ratchet_detect(R, K)
25:    end if
26:    K ← keycache(O_K, Roff, NRATCHET)
27:    IsOkEntry ← verify entry R for K, O_K, L
28:    if not IsOkEntry then
29:       FAIL()
30:    end if
31:    O[R.L] ← O[R.L] + R.Dsz,
32:    O_K ← O_K + Csz
33:    Roff ← (Roff + 1) mod NRATCHET
34: end for
35: for each log file L do
36:    pop contiguous from O[L]
37:    if O[L].heap not empty then
38:       FAIL()
39:    end if
40: end for
41: if (O_K/Csz) mod NRATCHET ≠ 0 then
42:    FAIL()
43: end if
44: SUCCESS()
```

Algorithm 5 Pop contiguous algorithm

```
1: done ← false
2: while O[L].heap not empty and not done do
3:    R_aux ← O[L].heap.min
4:    if O[L] ≠ R_aux.Loff then
5:       push R_aux onto O[L].heap
6:       done ← true
7:    else
8:       O[L] ← O[L] + R_aux.Dsz
9:    end if
10: end while
```

Algorithm 6 Ratchet detection algorithm

```
1: for NRATCHET in 1..MAXNRATCHET do
2:    K ← keycache(O_K, Roff, NRATCHET)
3:    IsOkEntry ←verify entry R for K, O_K, L
4:    if IsOkEntry then
5:       return NRATCHET
6:    end if
7: end for
8: FAIL()
```

Algorithm 7 Key cache algorithm

```
1: K, O_Kprev, Roff_prev ← last stored K, O_K, Roff
2: if O_K = O_Kprev and Roff = Roff_prev then
3:    return K :HIT
4: end if
5: if O_K ≠ O_Kprev or Roff_prev > Roff then
6:    K ← K_β[O_K ... O_K + Csz − 1] :REKEY
7:    Roff_prev ← 0
8: end if
9: if NRACHET ≠ 1 then
10:   K ← ratchet(K : Roff_prev → Roff, NRATCHET)
11: end if
12: store K, O_K, Roff :LOAD
13: return K
```

Algorithm 8 Verification algorithm for an entry

```
1: MD ← R.L||R.Loff||R.Dsz||R.Coff||R.Roff
2: D ← R.L[R.Loff ... R.Loff + R.Dsz − 1]
3: H′ = HMAC(K, MD||D)
4: if H′ ≠ R.H then
5:    FAIL()
6: end if
```

heaps are checked. If any heap is not empty, the verification fails because some areas of the log files have not been covered or the attacker inserted fake records in $SEAL_{log}$.

The size of the heap defines the level of disorder permitted for the verification. The heap must be big enough to support the possible overruns produced by the synchronization of concurrent processes performing append operations for a log file L. Note that an out of order append is, in general, a very improbable event in a real scenario.

Modular log verification (i.e., verifying only a portion of L) could be done similarly to complete verification, as in SealFSv1.

3.5 Ratcheting

In general, when using a pure ratchet, the entropy of the system is limited to the size of the initial secret or seed, because the security of the chain degrades linearly with the number of iterations (i.e., *epochs*) [1]. We combine ratcheting and rekeying from disk and burning the piece of the keystream to provide us with the necessary forgetful (non-reversible) process. This makes it theoretically more secure even if it uses more disk space. In SealFSv2, space can be exchanged for security by using different values of $NRATCHET$.

Setting $NRATCHET = 1$, SealFSv2 behaves as SealFSv1[6]. Setting a greater value for $NRATCHET$ moves the needle in the other direction with smaller demands on disk usage in exchange for linearly degrading the entropy inside the ratchet chunk. Every $NRATCHET$ chunk we reseed the entropy of the secret, which degrades linearly.

[6] Taking into account that the new version is performance tuned.

Theoretically, security increases with entropy. Given a perfect random number generator[7] to create the keystream, the entropy of the keys would be maximum if $NRATCHET = 1$. In practice, there is a trade-off between practical security and resource usage (space and time). How many resources are expended should be the result of a risk/benefit analysis. SealFSv2 permits to adjust the level of entropy with the $NRATCHET$ parameter and leaves this analysis to the user (which must define the threat model, etc.).

Our scheme provides almost random access to the keystream (taking in account each access may have to ratchet the key for that position $NRATCHET$ times or less) in verification time; we do not need to calculate all the keys, only the offset from the last rekeying. Pure 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.

Ratcheting introduces some interesting issues. When the key is ratcheted, no chunk is burnt from the keystream. In order to preserve the guarantee that L cannot be truncated without detection (given by line 13 in Algorithm 4), we have to *burn fake entries* in $SEAL_{log}$ when the system stops. Once the system is stopped, the number of entries in $SEAL_{log}$ will be an integer multiple of $NRATCHET$.

These entries correspond to a special log file which does not exist[8]. If $SEAL_{log}$ does not contain $NRATCHET * n$ number of entries, then it has been tampered with. Else, the log is correctly *sealed*.

Another important detail is that the value of $NRATCHET$ is not written anywhere, but it is used to derive the keys. In order to recover the value of $NRATCHET$ in the verification, different values are tested on the first entry until the HMAC matches. We call this process *ratchet detection* (see Algorithm 6).

In our original paper [7], we provided some anecdotal evidence of the service that could be provided by $32GB$ under normal usage of the system (14 years) and under heavy usage (like a Hadoop Server: 1.2 years). Ratcheting the key with $NRATCHET = 64$ already scales any of this servers well beyond its usable lifespan even while taking into account that keys in SealFSv2 are 32 bytes long and in SealFSv1 occupied 20 bytes which gives a lifespan of they keystream of 560 years and 48 years, respectively. Given that we change the key every 4 years, we could do for the worst case (the server with heavy usage) with $3GB$. These smaller sizes make key generation and managing simpler and faster. How much should our could $NRATCHET$ grow? It depends on two factors: How much disk is available for $SEAL_{log}$ (remember that stopping the system may cost some $NRATCHET$ log entries) and how much theoretical security we want to provide. The bigger the $NRATCHET$, the more we spread the entropy of the original keystream chunk.

4 SealFSv2 implementation

The implementation of SealFSv1 is described elsewhere [7]. It is a stackable file system, which is mounted on top of another filesystem to provide extra functionality. The file system is implemented as a Linux kernel module. The implementation of SealFSv2 is an evolution of the original filesystem. It serves the same objects (files and directories) as the underlying directory (the mount point) and hooks the write operations that append data to files in order to execute the write algorithm described in the previous section. Note that there can be independent instances of SealFS mounted in separate points if needed to isolate applications.

The authentication procedure described by Algorithm 1 is transparent for the different concurrent user space processes: they manage log files as usual (by opening the log file and performing append `write` system calls). Like the previous version:

- It uses the *i-node* number to identify files (this is important for log rotation by renaming). It provides the same numbers than the underlying filesystem. Some times, SealFSv2 uses fake, invalid i-node numbers for the fake records used to *seal* $SEAL_{log}$ when needed (as explained in Sect. 3.5). A problem of using i-nodes as identifiers for files appears if the files are copied somewhere else for verification. In this case, their i-node numbers will change. To this effect, the new verification tool accepts a list of i-node number mappings (from the original number to the number in the new filesystem) so the verification can still be done.
- The current implementation uses the HMAC-SHA-256 [36] algorithm to authenticate the logging data. The key length used for the algorithm is 32 bytes (independently of the value of $NRATCHET$).

4.1 Concurrent writes

When a user process performs a `write` system call, the SealFSv2 hook is called (in kernel space). Then, the hook function has to execute the write algorithm. It performs the write to 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 and calculates the real offset for this write operation (current size minus the number of bytes returned by the write operation).

[7] Note that we do not define or impose the source for the keystream; this is out of the scope of our work. We recommend to use a CSPRNG properly reseeded with specialized hardware.

[8] In our implementation, it has a special invalid i-node number as we will explain.

Two mechanisms are used for synchronization:

- To synchronize the processes, we use the lock stored in the i-node for each file. Write operations over different files can be executed concurrently.
- A global per mount point (i.e., SealFS instance) mutex is used to preserve the order of operations in $SEAL_{log}$ and protect the offsets in the header of K_α. After the append-only write is done and the real offset is known, the process acquires the global mutex to execute the blue block of algorithm 1. Once the key is calculated and the offset for the $SEAL_{log}$ is known, the global mutex is released and each process can proceed independently. Note that the global mutex is acquired after writing to the log file. Overruns described in Sect. 3.3.1 may happen here, after releasing the i-node lock and before acquiring the global mutex. It is not probable, but it is possible. We tolerate this race condition, which does not affect correctness, in order to avoid a bigger critical region (that should include the slow write operation). Such a big critical region would cause too much contention and performance issues. As explained before, this *benign* race condition is compensated by the verification algorithm.

4.2 Burning the keystream

When our system consumes a chunk of the keystream, the chunk needs to be *burnt*. In theory, overwritten data can be recovered through complex analysis performed in highly specialized laboratories [37]. In practice, for current high density magnetic disks, it suffices from 3 to 7 write passes with pseudorandom data [38]. Other kinds of storage devices (e.g. SSD) may be trickier. In this case, we should use the tools provided by the manufacturer. Thus, our implementation for keystream burning must be extensible.

From the standpoint of correctness and performance, how to coordinate the kernel and the processes (those performing the write system call) is another important synchronization issue. There are various prerequisites to be met:

- The costly burn process must not interfere with the performance of the user process by delaying the write operation response.
- The first burn must happen as early as possible to narrow the window of opportunity of an attack.
- An easy way to extend and implement new burning procedures (alternatives for pseudorandom overwrites) must be in place.

In SealFSv1, the first pseudorandom overwrite was done synchronously in the hook function. Thus, it is executed in the context of the user process performing the `write` system call. This had two problems with respect to performance:

- The user process had to wait for it to finish to return to user space.
- The operations could not be batched when there were multiple processes writing at the same time.

To alleviate these problems, SealFSv2 does it asynchronously. It runs various kernel threads named `k_sealfsd` with the main objective of burning the keystream. These threads do not have user space and cannot be killed, so it is guaranteed that they will keep running until the file system is unmounted.

All `k_sealfsd` threads are woken up periodically and go to sleep when there is no work. We make the window for burning as small as possible by waking up one of them whenever a write is done. Whenever one of these threads is active, it will burn as much keystream as possible batching various regions together if the burnt offset advanced more than one time while it was sleeping.

These threads try to synchronize the portion burnt of the keystream as much as possible within the Linux kernel (calling `vfs_fsync_range` which is not necessarily honored by all filesystems, but is the best that can be done).

After the woken up thread burns that portion, the other kernel threads will periodically try to *re-burn* the same region (once per thread). While one of the threads is woken up explicitly by the writing process, the others wake up at regular intervals. This could make them interfere if they happen to synchronize by chance. In order to keep that from happening, following a strategy inspired by cicadas[9], the threads will wake up at intervals which are relatively prime, lowering the number of collisions even in the presence of some random interference between threads.

In order to keep the kernel threads abreast of the portion that needs to be burnt (and can be burnt), they share an integer using atomic integer operations on a shared variable. This variable is different for each SealFSv2 instance (i.e., mount point). This is done in order to keep the interference minimal and keep the kernel threads running independently as much as possible. Given that the only communication between these kernel threads happens through an atomic variable, the semantics of the memory model of the Linux kernel has to be carefully considered. Note that if the semantics are relaxed enough (like those in C11 [39]), the read from the keystream could be performed after the key is already burnt. In our case, it cannot happen, because the Linux kernel documentation[10] states explicitly that *"read-modify-write (RMW) operations that have a return value are fully ordered"* and *"fully ordered primitives are ordered against everything prior and everything subsequent."*

[9] Insects of the genus *Magicicada* with long life cycles defined by prime numbers.

[10] `/Documentation/atomic.txt` file in the Linux kernel tree.

In other words, an atomic operation which returns a value behaves as if a full memory barrier was executed before and after the operation.

After the first write by the woken kernel thread, it is considerably difficult to recover the burnt portion without a very specialized lower layer attack. Note that the other kernel threads will also rewrite the portion later.

Apart from the `k_sealfsd` threads, there is a user space service (in Unix terminology, a *daemon*) named `sealfsd`. This daemon can read the current offset from the file header ($K_\alpha header.off$), which is kept synchronized by the kernel threads. Then, it can perform extra burning actions by implementing further deleting strategies. Being in user space, it can be easily modified. Thus, it covers the extensibility requirement discussed before. Note that the blocks being written by the `k_sealfsd` threads and the `sealfsd` daemon are not accessed by our 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[11].

Keeping this integer (the offset) advancing uniformly without holes, requires making the update of the integer and the read of the key an atomic operation, which means keeping a lock acquired for the clients. More complex strategies can make this more efficient in theory (for example, by using a more complex data structure to keep the pieces which are burnt in a non-contiguous way). This would mean more interference with the kernel threads, which would either share this data structure (with more synchronization problems) or send individual operations through them.

We tried some of these methods and they ended being similar or worse in performance while introducing significant complexity and memory management problems. We also tried simply using a lock but, while simpler, the contention makes performance drop (remember this would be one lock per mount point). This last strategy is the one we followed in SealFSv1, which performs worse than SealFSv2. Note that SealFSv2 also includes other optimizations. For example, to make the writes faster and not allocate memory and copy the content of the user buffer to it, it maps the corresponding pages and hashes the user memory directly, which is considerably faster (not only do we save the copy and pollution of the cache, but the contention when allocating the memory).

The solution of a uniformly advancing integer appears to be the best compromise between simplicity and performance. It also enables the `sealfsd` daemon to use the same strategy (by reading the integer from the header of the log, which is updated periodically by the burner threads). Another strategy which may help is to use a raw device (a partition or a raw hard disk) to keep the keystream. This has some advantages and some disadvantages. First, it gives us finer control of how many copies are in the disk (discounting the extra ones make by the vendor disk firmware). It reduces the interference of the filesystem. In exchange, we loose the consistency protection offered by the filesystem, specially for the metadata (the journal and so on) in case of a loss of power.

[11] Note that a disk block holds 16 keys in the current implementation. Re-burning the last portions of keystream only requires rewriting a few blocks.

4.3 Ratcheting

One important detail to take into account is that the ratcheting process is stateful. The last key ratcheted is kept in memory. Upon reboot, it would be lost. The state on disk is the keystream. In order to be able to unmount and reboot gracefully, we add some extra log entries (with length zero and a special invalid inode) in order to consume the rest of the ratcheting process. This way, when the first write after a mount occurs, it provokes a rekey, and it can continue normally.

Unmounting the system triggers the sealing process. This way, it is guaranteed that there is a one-to-one mapping between the index of each log entry and the tuple $< key, ratchet, offset >$. Take into account that this wastes $NRATCHET$ log entries per unmount, which can be significant if $NRATCHET$ is big.

A way to alleviate this would be to write only the first entry of the seal and consider that one to represent $NRATCHET$ entries and advance the offset for the first write after the next mount. Then, the filesystem would leave a hole of the correct size with zeroes. Many filesystems have support for holes, so the seal would occupy in them almost no real disk size (while keeping the mapping between keystream and offset).

4.4 Tools

There are other user space tools, like in SealFSv1 [7]. The `dump` command lists all the records of $SEAL_{log}$. The `prep` command creates the keystreams K_α and K_β by reading data from a the secure pseudorandom data generator. Note that `prep` does not know anything about $NRATCHET$; it is a parameter for `mount`. Both commands are similar to the ones in SealFSv1. The `verify` command implements the verification algorithm described in Sect. 3.4. In addition, the new version accepts parameters to map i-node numbers (needed to verify the files in a different Ext4 filesystem).

4.5 Testing

We have used a machine emulator and virtualizer, QEMU [40], and a minimalistic Go userspace, U-root [41], to build a testing environment. This environment is easy to understand and highly maintainable. It allows us to automatically run a set

of end-to-end regression tests. It has worked very well for testing and preserving the sanity of the developers.

In this environment, QEMU starts a Linux system that mounts the SealFSv2 filesystems, runs the tests, and unmount them, without any interaction with the user. Thanks to the minimalistic nature of U-root, this can be done in less than 350 lines of simple shell scripting code.

5 Evaluation

In order to perform a complete evaluation of the system, we have conducted two different experiments using:

1. A microbenchmark similar to the one described in [7], where we used the timestamp counters (TSC) of the cores from a user space process to measure the time in cycles elapsed during a write system call on a SealFSv2 mounted over an Ext4 filesystem.
2. A standard benchmark, *Filebench* [8], also for a SealFSv2 mounted over an Ext4 filesystem.

The machine we have measured it in is a 1.60GHz Intel Core i5-10210U CPU with 6144 KB of cache, 16 GB of RAM and an SSD hard disk model HFM512GDJTNI-82A0A. The machine runs Ubuntu Linux 20.04 with a 5.4.0 x86_64 Linux kernel.

5.1 Custom microbenchmark

Unless explicitly stated otherwise, measurements have been taken at least 1000 times and then the median value obtained. Before each measurement round, all the caches of the system are dropped to minimize interference and some writes are executed to preheat the cache.

We have measured different versions of the kernel module:

- `vers_PASS`: A version which is a pass through filesystem. In this version, the overhead is caused only by traversing our intermediate filesystem (with all the rest of the code commented).
- `vers_NOHMAC`: A version without the HMAC calculation (the HMAC code is commented in this case). Note that this version does not strictly work: The verification will fail. It is only used for measuring against the regular version with different number of processes and for various values of $NRATCHET$.
- *`vers_NRATCHET N`*: A regular complete version with several values N for $NRATCHET$.
- `vers_NORATCHET`: A version of the kernel with the ratchet code commented. Note that $vers\_RATCHET1$ is the just the regular version with $NRATCHET = 1$ (i.e., no ratcheting). It is logically equivalent to `vers_NORATCHET`, and should be functionally equivalent, but it is not. The reason is that $vers\_RATCHET1$ has a conditional jump that controls if it has to do the ratcheting or not. For performance reasons, this conditional jump is annotated to hint the branch unit the wrong way: it is biased toward not rekeying (normally, with any value of $NRATCHET > 2$ this is a good idea). The penalty for this wrong speculation is big for that case.

Note that the colors in the microbenchmark graphs correspond with the numbers derived as we explain next. We can estimate by calculation the approximate cost of each part of the final code by subtracting the measurements of the different versions to the baseline version: a basic naked Ext4 filesystem (labeled as `ext` in the graphs). For example, the median cost of a write in SealFSv2 configured with $NRATCHET = 4$ can be decomposed as:

- Cost of the Virtual Filesystem Switch (VFS) and bookkeeping of the stacked filesystem:

$$OVERLAY = vers\_PASS$$

- Cost of computing the HMAC of the user data and log entry:

$$HMAC = vers\_NORATCHET - vers\_NOHMAC$$

- Cost of ratcheting the key:

$$RATCHET = vers\_RATCHET4 - vers\_NORATCHET$$

- Cost of reading a new key and general key bookkeeping:

$$REKEYING = vers\_RATCHET4 - RATCHET - HMAC - OVERLAY$$

We ran the experiments for different values of $NRATCHET$ (labeled as `ratchet001`, `ratchet002`, etc. in the graphs). We also ran the experiments with processes sharing the file descriptors for open files (i.e., inheriting them already open, suffix `-s` in the labels) and without sharing them (i.e., opening them after creating the process, suffix `-p` in the labels). Note that the Linux kernel acquires a lock when the file descriptor is shared[12] in order to protect the file descriptor offset from race conditions.

We first ran the experiments with a write size of 100 bytes (which is around what is normally written in a log, see [7] for the rationale) and for different number of processes and with and without shared file descriptors. Later, we ran the

[12] See `file.c`, the `__fdget_pos` function in the Linux kernel.

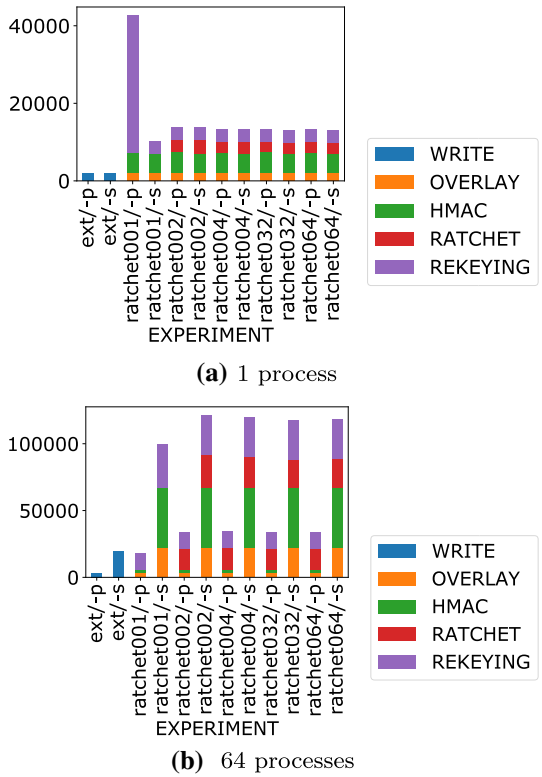


Fig. 2 Ext4 versus different SealFSv2 configurations, with shared (-s) and non-shared file descriptors (-p)

experiments with different write sizes for non shared file descriptors, 1 and 64 processes.

In Figs. 2 and 3, we depict the results of the calculations described above. Note that writing to the filesystem happens asynchronously (through the file cache), so the cost we measure for Ext4 is almost the cost of a memory copy and some synchronization, which is extraordinarily fast.

Figure 2 represents all the measurements together grouped by the number of processes in the benchmark and provides an overview of the different costs. The first thing that becomes apparent is that sharing the file descriptor has an unreasonable cost, both for the regular Ext4 filesystem and for SealFS. In these case (-s), all writes are mutually exclusive (because of the file descriptor offset lock in the Linux kernel, as explained before). Nevertheless, it is an interesting measurement to understand the cost of the whole operation discounting the gains obtained by parallelization and concurrency.

More detail and in-depth analysis can be obtained from Fig. 3. Here we can see that, when the file descriptors are not shared, some of the costs are hidden by parallelization and concurrency. Note that this does not happen for the cost of the ratchet and happens only partly for rekeying. The problem is that, in order for one write to progress, it has to use a key that depends on the previous key. Thus, there is a serialization effect.

When there was no ratcheting, the key could be read in parallel. On the other hand, for ratcheting, we must block on a lock, waiting for the previous key to be ratcheted serializing the operations.

In any case, considering that the cost for the baseline (`ext`, the naked Ext4 version) is extremely fast. The worst case for our measurements in realistic conditions is less than 8 times of the naked version; it's perfectly useable for logging.

We pay the cost for the ratcheting (red in the graphics) in exchange for making the keystream smaller. Note that for one process (or, equivalently, when the file descriptors are shared), most of the cost comes from the HMAC. Another cost to take into account is the space used when unmounting the device ($NRATCHET$ entries).

Note that, for $NRATCHET = 1$, results are quite bad in Figs. 2a and 3b. The branch annotation effect illustrated previously explains these bad results. It is not so much of a performance bottleneck to be a real problem. Nevertheless, if needed, it could be easily solved in a further production implementation (e.g., by registering a pointer to a function without the ratchet implementation when the system is mounted and $NRATCHET = 1$).

The measurements for different write sizes can be seen in Fig. 4. Big write sizes are not representative. Normally, logs are written one or several lines at a time. Nevertheless, these measurements are still interesting to understand the cost of the $HMAC$ and the behavior of the filesystem in such cases. Performance is slightly worse, around ten times (10 times) the reference naked Ext4 filesystem. As the write size grows, most of the cycles are taken by the HMAC calculation and writing to the underlying filesystem.

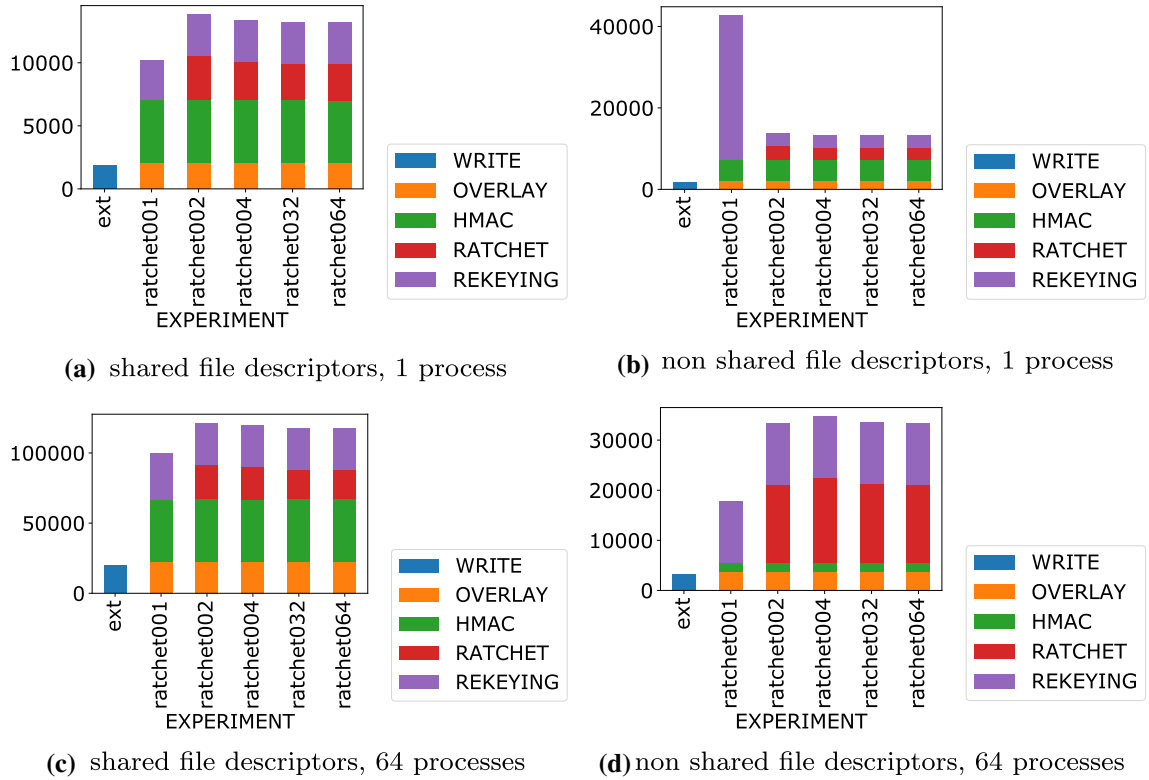
As shown by the experiments, we conducted for SealFSv1 [7], as expected, different instances of our filesystems (i.e., concurrent filesystems using different mount points) are independent (because they do not share any internal resource). Thus, it is not worth measuring this case again.

5.2 Standard benchmark: Filebench

We also used a standard benchmark for files, Filebench [8], to evaluate SealFSv2. In order to use Filebench, we measured a slightly modified version of the filesystem which considered any open operation on the filesystem to be append only in any case. Note that the cost of this modification is negligible.

As Filebench measures operations per second and conversely bandwidth, the results are better than for the micro-benchmark. Here, the filesystem exploits the concurrency of its implementation and the latency is hidden by the fastest operations. The results, taking this amortized cost in account, are quite good and can be seen in Figs. 5, 6, 7 and 8.

We have measured values of $NRATCHET$ up to 1048576, to see how much the performance degrades in that extreme case (that simulates the pure ratchet approach). It



(a) shared file descriptors, 1 process

(b) non shared file descriptors, 1 process

(c) shared file descriptors, 64 processes

(d) non shared file descriptors, 64 processes

Fig. 3 Ext4 versus different SealFSv2 configurations, with shared (-s) and non-shared file descriptors (-p)

is still less than 8 times of amortized cost, which would be usable. Take into account that this value for $NRATCHET$ would not be advisable, because the cost for sealing the ratchet (unmounting and mounting again) would be for the worst case 72 MB (1048576 entries at 72 bytes per entry).

5.3 Performance

As explained in our previous work, SealFSv1 [7] was expensive, it was not suitable for intensive I/O. Nevertheless, it was fast enough to secure regular log files (which do not require intensive I/O).

SealFSv2 is optimized, so it performs better than SealFSv1. Nevertheless, it is not suitable for intensive I/O either. In short, the Filebench bandwidth results (Figs. 6 and 8) show that the penalty for using SealFSv2 is approximately between 1/3 and 2/3 (compared to Ext4), depending on the concurrency level. This cost seems reasonable for tamper-evident logs.

Nevertheless, there is still room for improvement. The microbenchmark results (used to itemize the costs) show that the times to compute the HMAC (green bar in the microbenchmark figures) and the ratcheting procedure (the red bar) are noticeable. We could make SealFSv2 faster by using other lightweight cryptographic algorithms, without sacrificing the key features of our system: (i) rekeying with a precomputed keystream to add entropy (the purple bar) and (ii) the stackable filesystem implementation (the orange bar), a transparent way to protect the logs of existing software without requiring any modification of applications, libraries, and frameworks.

5.4 Verification times

Median verification times for 128 measurements are depicted in Fig. 9. Note that using random offsets does not incur in any extra cost. The cost of using the ratchet is roughly 1.5 times cost of not using it and is constant for any size.

5.5 Discussion: storage-based versus ratchet versus hybrid schemes

As explained, the verification for the storage-based approach is faster (approximately 0.7 times the cost) than for hybrid approaches (and therefore the pure ratchet approach).

Regarding the filesystem operation (i.e., appending data to the log files), the storage-based approach seems to perform better than hybrid approaches (only slightly better in most cases). In general, this difference may not be significant to conclude that the pure storage-based approach performs bet-

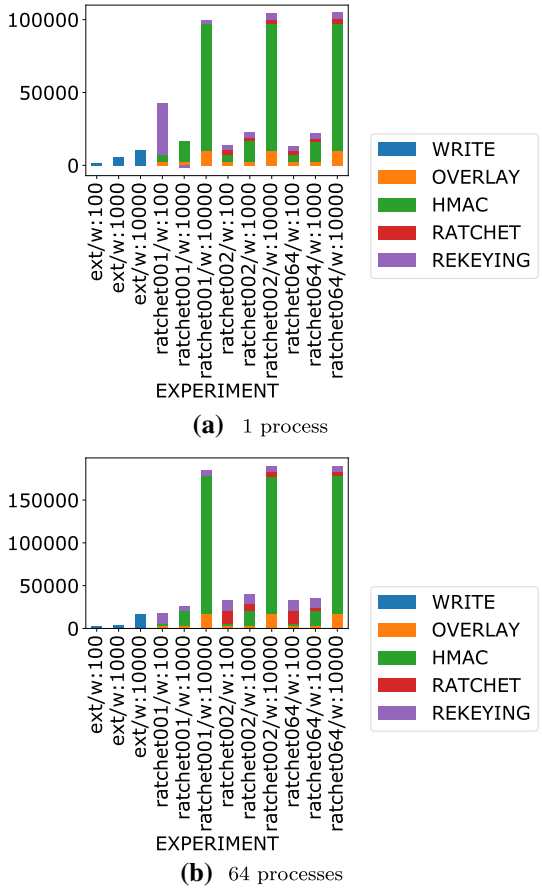


Fig. 4 Ext4 versus different SealFSv2 configurations, non-shared file descriptors, for different write sizes

ter than the hybrid approaches with $NRATCHET \leq 64$. Little details in the implementation could decide the winner. The branch annotation issue for $NRATCHET = 1$ is a proof: in this case, an apparently innocent fine-grained implementation detail triggers an architectural effect (for an specific kind of machine, Intel) that makes the pure storage-based approach worse than the hybrid approach.

The pure ratchet approach (simulated by $NRATCHET = 1048576$) performs way worse than the pure storage-based approach (approximately 1.6 times the cost, as shown in Fig. 7).

We did not expect these results: we suspected that the ratchet approach could outperform the storage-based approach (CPU bound vs. I/O bound). Sometimes, experimental results are surprising.

The results can be explained by the important optimizations for storage operations present in modern operating systems like Linux (read-ahead and so on). It becomes apparent that the storage-based approach may be more suitable for current systems than the ratchet approach.

In any event, in the authors opinion, performance should not be the key factor to choose the storage-based approach over the ratchet approach. In both cases, the performance is good enough for logging. The trade-off is between storage space and degrading security. Another element to take into account when considering the keystream disk size is the time taken to generate it, copy it, etc. A smaller keystream will be in general more manageable. This could also tilt the balance to a hybrid approach.

5.6 Limitations of the study

This study has some limitations.

5.6.1 Entropy loss quantification

While Bellare stated that a pure ratchet degrades linearly [1] and there is a consensus that entropy loss is a defect [42–45], it is difficult to quantify how much entropy is enough to keep a system secure. Would a pure ratchet initialized with a 512-bit secret be secure enough for most applications? This is a difficult theoretical question we do not try to answer in this study, and thus, a limitation. Answering that question would require concrete scenarios with threat models which quantify the computational power of the actors. In addition, it requires developing a sophisticated theoretical cryptoanalytical framework to specify entropy loss-based attacks.

Due to time and space constraints, we have not tried to perform this analysis, which would allow the user to quantify better the security benefit of adding more or less entropy. Such analysis would guide a practitioner to find the best possible configuration (i.e., the value of the `NRATCHET` parameter in SealFSv2). The only theoretical analysis we know of is the estimations given by Bellare [1] we already mentioned.

Finding an optimum amount of entropy, and thus the smallest keystream necessary to keep the system safe in a particular scenario, is left as an open problem that can be addressed in future research.

5.6.2 Quantitative performance comparison

This study does not include a performance comparison with previous systems cited in Sect. 2, such as Logcrypt, I^{3}FS, BBR, CUSTOS, KennyLoggings, etc. Note that those systems follow dissimilar approaches and provide different features. Moreover, the numbers presented in those works correspond to disparate implementations, cryptographic algorithms, environments, benchmarks, operating systems, and hardware. A fair quantitative comparison, when possible (note that some of them do not provide exactly the same functionality), requires rigorous measurements on the same setup.

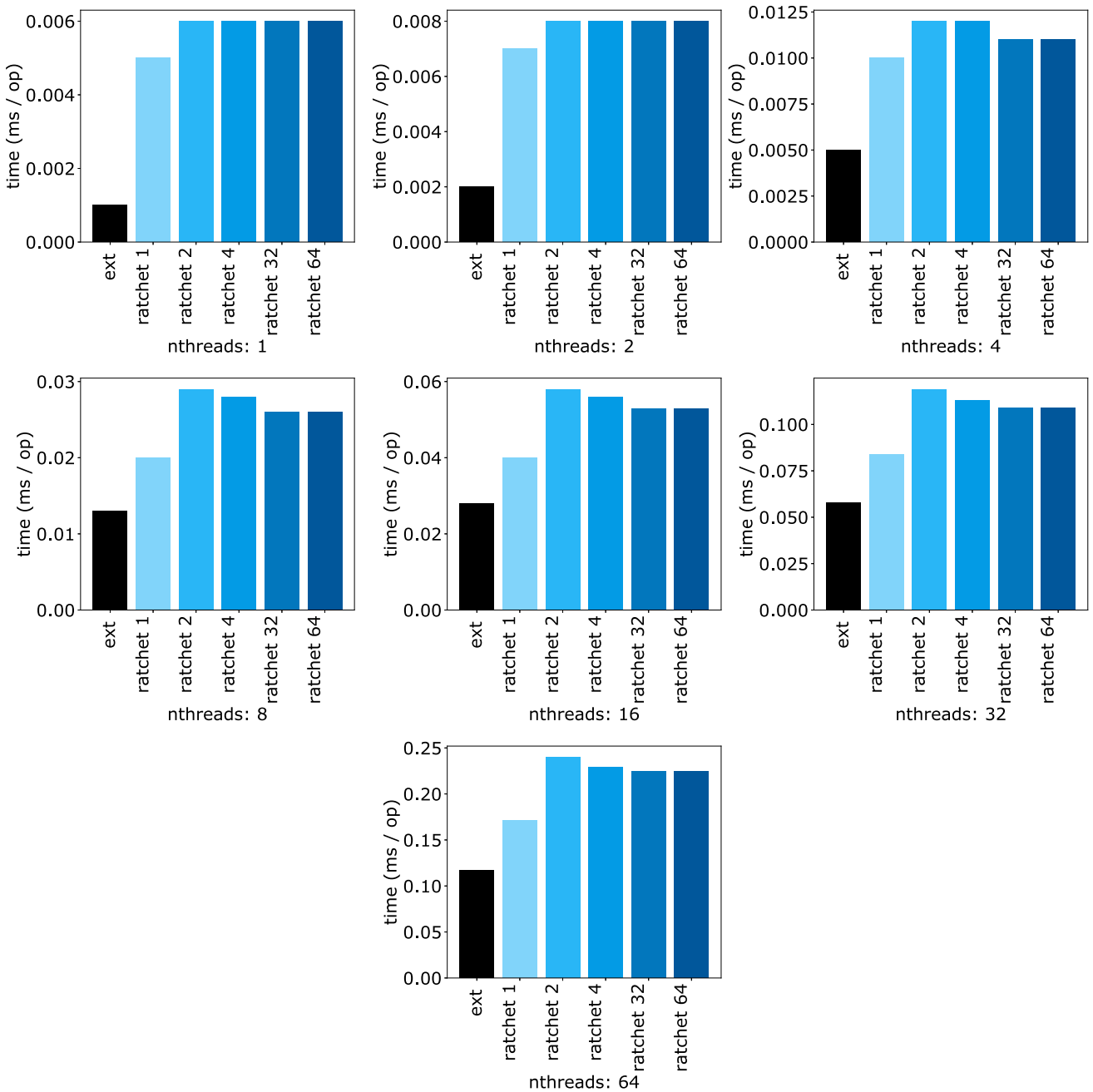


Fig. 5 Filebench operations per second

Due to time and material constraints (e.g., access to the original software and configurations), we have not tried to conduct this experimental analysis.

6 Conclusions

We have described SealFSv2, a new version of local tamper-evident logging system focused on the forward integrity model. This new version mixes two opposite approaches: our original storage-based approach and the classical ratchet approach.

The paper provides: (i) a detailed description of the new version and its algorithms; (ii) fine-grained details of a fully functional open-source implementation, a Linux kernel module for a stackable file system; (iii) our experience building this system; (iv) and a complete evaluation with a custom minibenchmark and a standard benchmark for file systems, which permits to compare the performance of the two confronted approaches (and intermediate ones). The

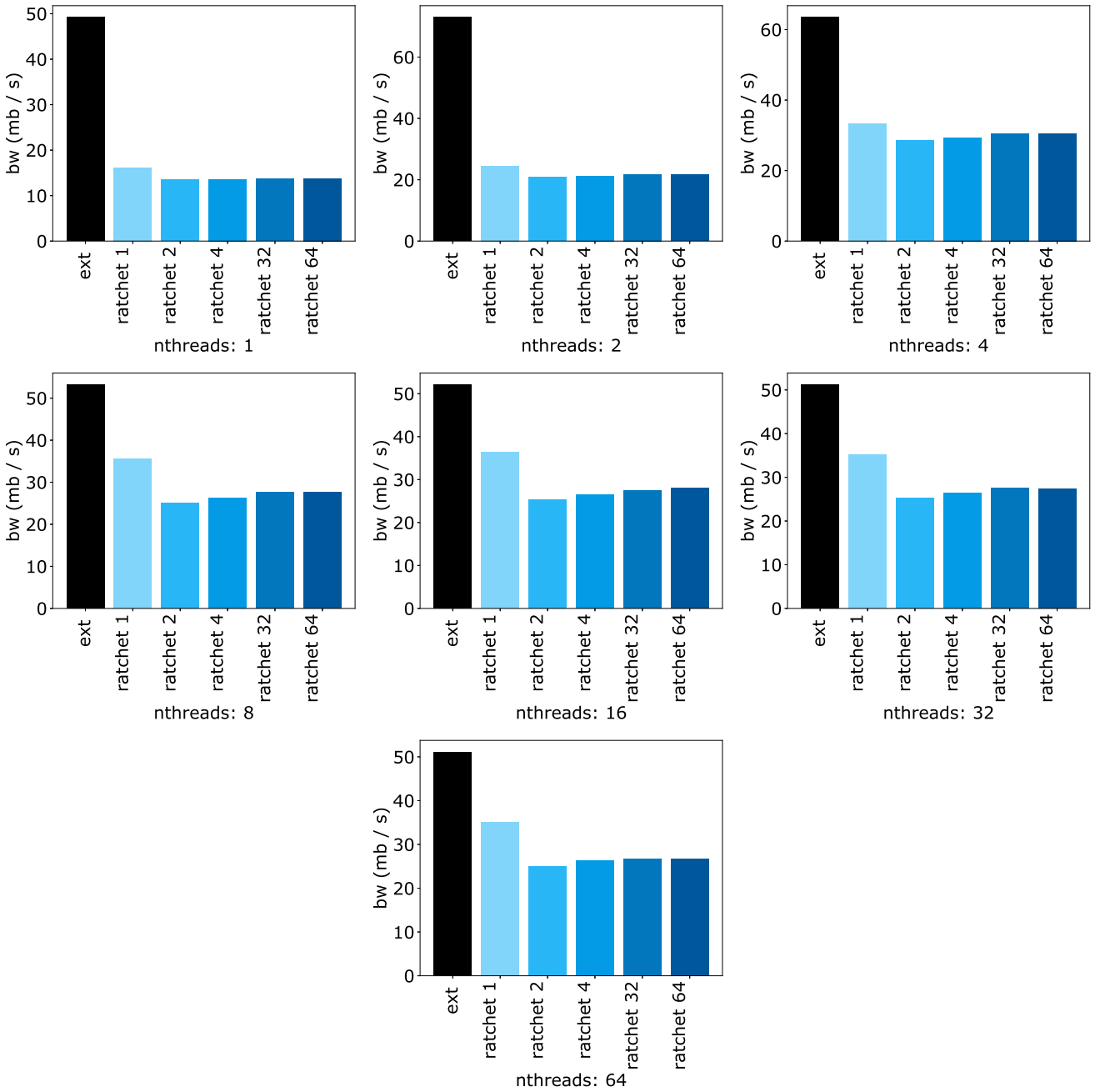


Fig. 6 Filebench bandwidth

experimental results suggest that the storage-based approach is more suitable for current operating systems running on general-purpose hardware.

Other than dealing with the limitations explained in Sect. 5.6, future work for SealFS includes disposing of the external drive by encrypting the key with a public key (or an epoch session key which could be then saved encrypted) instead of burning it. Going further, the key could be generated dynamically from the kernel entropy pool (i.e., a device like `/dev/random`) or a special device and saved encrypted.

Further future work includes configurable cryptographic algorithms for authentication and ratcheting (including lightweight cryptographic algorithms to improve performance), dynamic variations of the ratchet parameters for partial security degradation, signed keystreams for third-

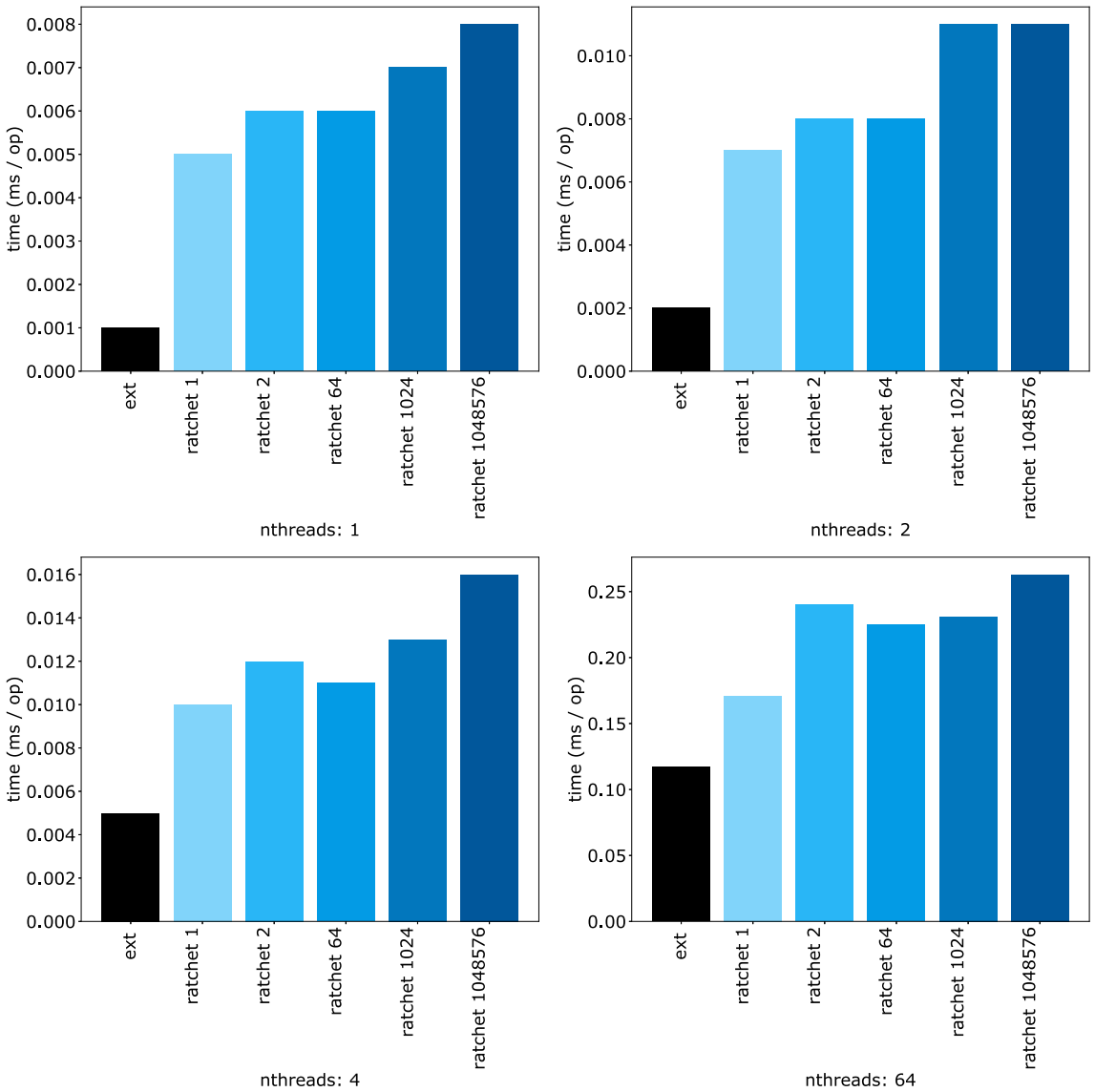


Fig. 7 Filebench operations per second including bigger $NRATCHET$

party verification and an implementation of the SealFSv2 approach for robotic middleware[13].

The source code of SealFSv2 can be downloaded from: https://gitlab.etsit.urjc.es/esoriano/sealfs/tree/master

[13] Note that it is not feasible to replace or modify the kernel of some robotic systems.

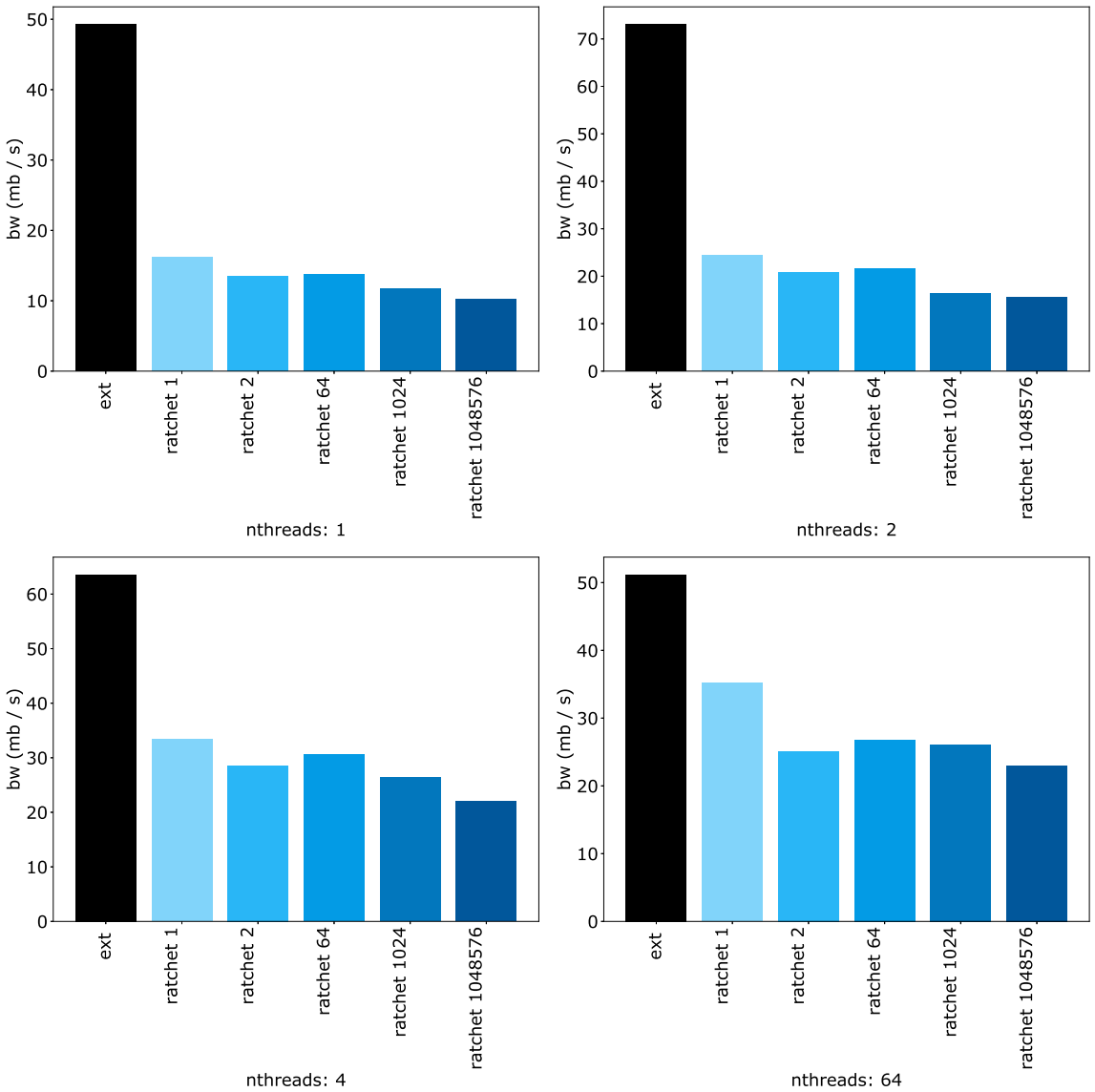
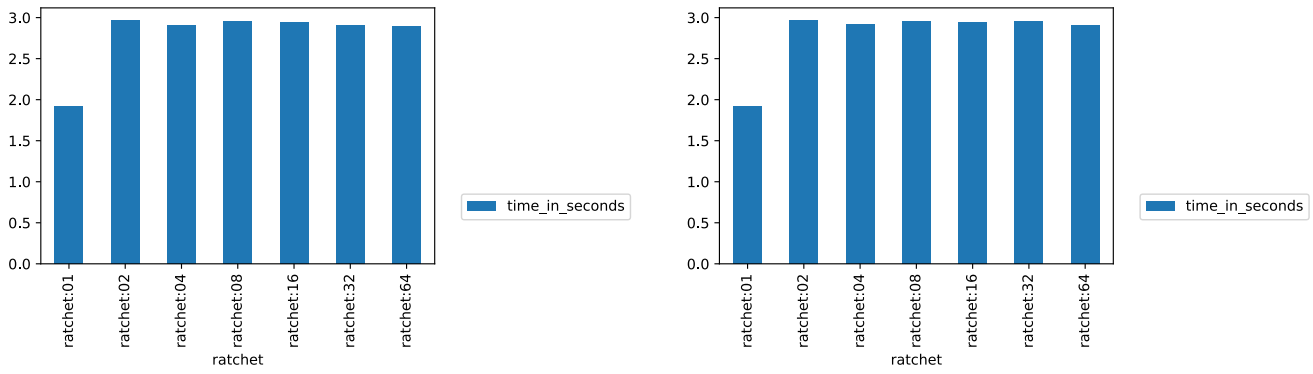


Fig. 8 Filebench bandwidth including bigger $NRATCHET$



(a) Sequential **(b)** Random offsets

Fig. 9 Verification times for 1000000 entries of write size 100 bytes and different NRATCHET

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work is partially funded under the Proyectos de Generación de Conocimiento 2021 call of Ministry of Science and Innovation of Spain co-funded by the European Union, project PID2021-126592OB-C22 CASCAR/DMARCE.

Data Availability Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

References

1. Bellare, M., Yee, B.S.: Forward Integrity for Secure Audit Logs. University of California at San Diego, Tech. Rep. (1997)
2. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology—CRYPTO 2017, pp. 619–650. Springer International Publishing, Cham (2017)
3. Cohn-Gordon, K., Cremers, C., Garratt, L.: On post-compromise security. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), June, pp. 164–178 (2016)
4. Schneier, B., Kelsey, J.: 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. [Online]. http://dl.acm.org/citation.cfm?id=1267549.1267553
5. Kelsey, J., Schneier, B.: Minimizing bandwidth for remote access to cryptographically protected audit logs. In: Recent Advances in Intrusion Detection, pp. 9 (1999)
6. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Trans. Inf. Syst. Secur. **2**(2), 159–176 (1999). https://doi.org/10.1145/317087.317089
7. Soriano-Salvador, E., Guardiola-Múzquiz, G.: Sealfs: storage-based tamper-evident logging. Comput. Secur. **108**, 102325 (2021)
8. Tarasov, V., Zadok, E., Shepler, S.: Filebench: a flexible framework for file system benchmarking. USENIX; Login **41**(1), 6–12 (2016)
9. Zeng, L., Chen, H., Xiao, Y.: Accountable administration and implementation in operating systems. In: 2011 IEEE Global Telecommunications Conference—GLOBECOM 2011, Dec, pp. 1–5 (2011)
10. Patil, S., Kashyap, A., Sivathanu, G., Zadok, E.: 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, p. 67-78 (2004)
11. Chou, B., Tatara, K., Sakuraba, T., Hori, Y., Sakurai, K.: 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, pp. 421–426 (2008)
12. Loggly, "Loggly: Remote Logging Service," https://www.loggly.com/solution/remote-logging-service/, 2019, [Online; accessed may-2019]
13. Stackdriver, "Stackdriver Logging," https://cloud.google.com/logging/, 2019, [Online; accessed may-2019]
14. Strunk, J. D., Goodson, G. R., Scheinholtz, M. L., Soules, C. A. N., Ganger, G. R.: "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)
15. Crosby, S. A., Wallach, D. S.: "Efficient data structures for tamper-evident logging. In: Proceedings of the 18th Conference on USENIX Security Symposium, ser. SSYM'09. USA: USENIX Association, p. 317-334 (2009)
16. Pulls, T., Peeters, R.: Balloon: A forward-secure append-only persistent authenticated data structure. IACR Cryptology ePrint Archive, vol. 2015, p. 7. [Online]. Available: https://eprint.iacr.org/2015/007 (2015)
17. White, R., Caiazza, G., Cortesi, A., Cho, Y., Christensen, H.: Black block recorder: immutable black box logging for robots via blockchain. IEEE J. Robot. Autom. **4**, 3812–3819 (2019)
18. Rosa, M., Barraca, J.P., Rocha, N.P.: Logging integrity with blockchain structures. In: Rocha, Á., Adeli, H., Reis, L.P., Costanzo, S. (eds.) New Knowledge in Information Systems and Technologies, pp. 83–93. Springer International Publishing, Cham (2019)
19. Wang, H., Yang, D., Duan, N., Guo, Y., Zhang, L.: Medusa: Blockchain powered log storage system, In: 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS), 11, pp. 518–521 (2018)
20. LogSentinel, "," https://logsentinel.com/, 2019, [Online; accessed may-2019]
21. Guardtime, "Blockchain Backed Log Assurance," https://guardtime.com/solutions/blockchain-backed-log-assurance, 2019, [Online; accessed may-2019]
22. Holt, J., Seamons, K.: Logcrypt: Forward security and public verification for secure audit logs. In: IACR Cryptol. ePrint Arch., (2005)
23. Ma, D., Tsudik, G.: A new approach to secure logging. ACM Trans. Storage (2009). https://doi.org/10.1145/1502777.1502779
24. Yavuz, A., Ning, P., Reiter, M.: Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In: Financial Cryptography, (2012)
25. Yavuz, A.A., Ning, P.: Baf: an efficient publicly verifiable secure audit logging scheme for distributed systems. Ann. Comput. Secur. Appl. Confer. **2009**, 219–228 (2009)
26. Hartung, G., Kaidel, B., Koch, A., Koch, J., Hartmann, D.: Practical and robust secure logging from fault-tolerant sequential aggregate signatures. In ProvSec, (2017)
27. Hartung, G.: Attacks on secure logging schemes. IACR Cryptol. ePrint Arch. **2017**, 95 (2017)
28. Paccagnella, R., Liao, K., Tian, D., Bates, A.: Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks. New York, NY, USA: Association for Computing Machinery, 2020, p. 1551-1574. [Online]. Available: https://doi.org/10.1145/3372297.3417862
29. Ma, S., Zhai, J., Kwon, Y., Lee, K. H., Zhang, X., Ciocarlie, G., Gehani, A., Yegneswaran, V., Xu, D., Jha, S.: Kernel-Supported Cost-Effective audit logging for causality tracking, in 2018 USENIX Annual Technical Conference (USENIX ATC 18). Boston, MA: USENIX Association, Jul pp. 241–254. [Online] (2018). Available: https://www.usenix.org/conference/atc18/presentation/ma-shiqing
30. Sinha, A., Jia, L., England, P., Lorch, J.R.: Continuous tamper-proof logging using tpm 20. In: Holz, T., Ioannidis, S. (eds.) Trust and Trustworthy Computing, pp. 19–36. Springer International Publishing, Cham (2014)

31. Nguyen, H., Acharya, B., Ivanov, R., Haeberlen, A., Phan, L. T. X., Sokolsky, O., Walker, J., Weimer, J., Hanson, W., Lee, I.: Cloud-based secure logger for medical devices. In: 2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE), pp. 89–94 (2016)
32. Karande, V., Bauman, E., Lin, Z., Khan, L.: 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, p. 19-30. [Online]. Available: https://doi.org/10.1145/3052973.3053034 (2017)
33. Paccagnella, R., Datta, P., Hassan, W. U., Bates, A., Fletcher, C., Miller, A., Tian, D.: Custos: Practical tamper-evident auditing of operating systems using trusted execution, Network and Distributed System Security Symposium, Jan[Online] (2020). http://par.nsf.gov/biblio/10146530
34. Schneier, B.: Data and Goliath: The Hidden Battles to Capture Your Data and Control Your World, 1st edn. W. W. Norton Company (2015)
35. Dhillon, V., Metcalf, D., Hooper, M.: The Hyperledger Project, pp. 139–149. Apress, Berkeley, CA (2017)
36. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication, IETF, RFC 2104, Feb. [Online]. http://tools.ietf.org/rfc/rfc2104.txt (1997)
37. Gutmann, P.: 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, p. 8 (1996)
38. U.S. National industrial security program operating manual DoD 5220.22-M. United States Department of Defense National Industrial Security Program, (2006)
39. ISO, ISO/IEC 9899:2011 Information technology, Programming languages: C. Geneva, Switzerland: International Organization for Standardization, December (2011)
40. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ser. ATEC '05. USA: USENIX Association, p. 41 (2005)
41. Minnich, R. G., Mirtchovski, A.: U-root: A go-based, firmware embeddable root file system with on-demand compilation," in 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15), (2015), pp. 577–586. [Online]. https://github.com/u-root/u-root
42. Dörre, F., Klebanov, V.: Practical detection of entropy loss in pseudo-random number generators," ser. CCS '16. New York, NY, USA: Association for Computing Machinery, (2016). [Online]. https://doi.org/10.1145/2976749.2978369
43. Kelsey, J., Schneier, B., Ferguson, N.: Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator, In: Selected Areas in Cryptography, (1999)
44. Kaptchuk, G., Jois, T. M., Green, M., Rubin, A. D.: Meteor: Cryptographically secure steganography for realistic distributions. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, (2021), p. 1529-1548. [Online]. https://doi.org/10.1145/3460120.3484550
45. NIST, Recommendation for random number generation using deterministic random bit generators," Computer Security Resource Center, Tech. Rep. NIST Special Publication 800-90A Revision 1, (2015)