Security Analysis of Forward Secure Log Sealing in Journald

Felix Dörre$^{1,2[0009-0009-7244-7753]}$ and Astrid Ottenhues$^{1,2[0009-0007-3082-216X]}$

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] KASTEL Security Research Labs, Karlsruhe, Germany
`{felix.doerre, ottenhues}@kit.edu`

Abstract. This paper presents a security analysis of forward-secure log sealing in the *journald* logging system, which is installed by default in almost all modern Linux distributions. Forward-secure log sealing is a cryptographic technique used to ensure the integrity of past log entries even in the event of a full system compromise. We identify multiple security vulnerabilities in *journald* resulting from a gap between the model of the cryptographic primitives and their usage in a larger context. Our contribution is both theoretical and practical: As a practical contribution, we discovered attacks on the log sealing in *journald* and provide descriptions as well as implementations of the attacks. In particular one vulnerability allows to forge arbitrary logs for past entries without the validation tool noticing any problem. This finding completely breaks the security guarantee of log sealing. For all described vulnerabilities we provide patches, the two more serious ones are merged in systemd version 255. As a theoretical contribution, we provide formal definitions that capture the expected security properties of log sealing. We demonstrate our attacks on the vulnerable version of *journald* by showing how an attacker can defeat this security definition. Furthermore, we provide a modified version of the logging scheme which underlies the one in *journald* and prove that it satisfies our security definition. Since our patches have been merged, our logging scheme is the basis for the log sealing in *journald*. This work narrows the gap between theory and practice. It provides a practical example of the problems that can occur when applying cryptographic primitives to a complex real world system. It makes the logging implementation used in many Linux distributions more secure and demonstrates the importance of rigorous security analysis of cryptographic systems.

Keywords: Log Sealing · Secure Logging · Attacks · Security Analysis · Provable Security.

1 Introduction

System logs are a crucial tool for determining the impact and cause of a security breach. After a break-in (attempt), a system administrator can inspect log entries of past events to determine attack vectors and assess the impact of the attack. The conclusions can be used to derive an appropriate incident response, like resolving the exploited vulnerability, creating additional defensive measures or informing impacted data subjects, which might be a requirement imposed by law. For these investigations, the chronology of events can be vital to determine what happened. However, the significance of logs is also known to attackers who routinely cleanse logs to hide traces of their attacks. Therefore it is important to protect system logs even on a fully compromised system. The two main types of protection are using specialized hardware (such as write-only storage or an external logging server) and log sealing. Log sealing is a cryptographic mechanism to make tampering with logs evident. As it does not require additional hardware, setting up log sealing is easier than the other methods.

The two main publicly available and practically usable implementations of log sealing are those included in *syslog-ng* and the mechanism implemented in *journald*. Log sealing was introduced into *journald* back in systemd version 198, in 2012. As *journald* is installed by default on many Linux distributions, enabling sealing only requires setup of the sealing keys. For many cryptographic systems, designing the system and implementing it are two completely different challenges. While for a theoretic specification there are typically very precise and formally sound security definitions, those often lack when such a system is implemented in a real environment. For *journald* the whole security definition in the documentation is, that log sealing "may be used to protect journal files from unnoticed alteration"[3]. One goal of this paper is to take this high level security goal and derive a formal security definition that is generally applicable to logging schemes.

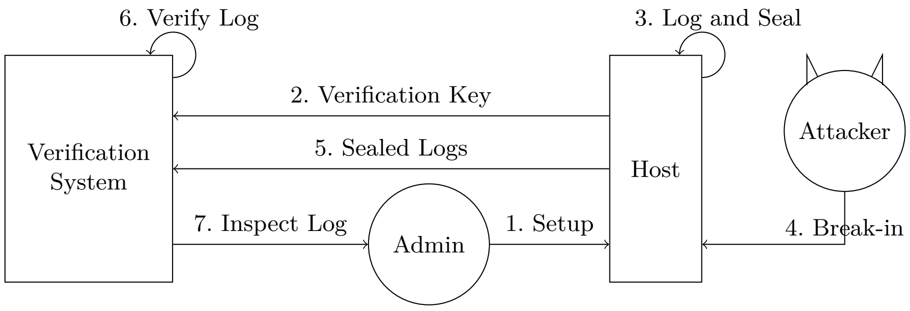

Fig. 1. A high level overview of how log sealing can detect log manipulation.

To give an idea of the security guarantee a user might expect and as an overview of the more formal definition we will give later, we now show the scenario where log sealing aims to make a difference (given as overview in Figure 1):

1. Sealing is enabled on a host system H.

[3] See `man journald.conf(5)`

2. The verification key is saved confidentially on a verification system V.
3. The system H logs application defined messages and seals every fixed time interval.
4. An attacker breaks into system H and has full access to the system. This includes reading the current sealing key and modifying all journal files.
5. When an administrator (auditor) suspects an attack, or also in regular, larger intervals, the log is copied from H to V.
6. The log is verified on V. Verification outputs both, whether the log is consistent and the time span for which verification succeeded.
7. On system V the administrator displays the log with various `journalctl` options to draw conclusions about the attack.

The intuitive security expectation would be, that an attacker can not modify (or delete/truncate) the journal on system H for all messages that were in a finished epoch before the break-in, without the verification raising an alarm.

The main idea of *journald* for solving this problem is depicted in Figure 2. The log messages are regularly authenticated with a key that is destroyed after usage and bound to a fixed real-time epoch. That way an attacker compromising the system later, cannot forge old log messages, as this would require authentication with a key that is already destroyed.

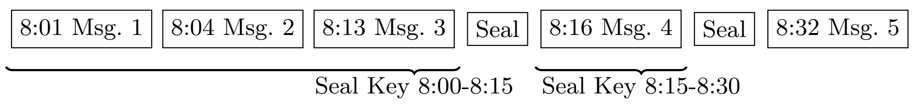

Fig. 2. A simplified example of a sealed log at the time "8:35" with sealing interval 15min. Messages 1-4 are sealed with the corresponding keys which have already been destroyed. The key for 8:30-8:45, has not been used, because that epoch has not ended yet. This key and all future keys are available to an attacker who would breach the system now.

1.1 Contribution

Our contribution is twofold. As our first main contribution, we uncover three practical security vulnerabilities, which all allow an attacker to break the expected security of the implemented log sealing by *journald*:

– Produce arbitrary logs for an arbitrary point of time in the past. This is resolved by verifying that old entries are not sealed with too recent key.
– Truncate the log. This is resolved by verifying that sealed log epochs are continuous.
– Hide some specific log entries when displaying the log with filters. We recommend to resolve this, by thoroughly verifying the fast access structures in the log file.

All these vulnerabilities allow modifying the displayed log content without the verification tool noticing any tampering. They all were present in *journald* since the original introduction of log sealing in 2012. Two of the patches we provided are merged in systemd version 255. To round this up, we provide a toolkit for inspecting and modifying journal files to have an environment where we can verify the described attacks to demonstrate their practical impact.

To provide more than just ad-hoc security guarantees, our second main contribution consists of a formal security model for log sealing and a logging scheme that is secure in this model. Our security definition is not fulfilled by the previous vulnerable implementation of *journald* as in Section 5 we show how two of the vulnerabilities translate to an attacker which wins the game of the security definition. However, our modified logging scheme does fulfill our security notion and is the basis of the log sealing in *journald*, since systemd version 255. This underlines the relevance of security analyses connecting theoretical specifications of a security promise to the real-world implementation of a whole system.

We bring theory and practice closer together by providing patches for the vulnerabilities in Section 5. The basis for these patches is the logging scheme which we present in Section 3 and which we show to satisfy our new security notion. Our second main contribution can be summarized as follows:

- We propose a formal security definition for forward secure log sealing `FS-EUF-CLMA` (Section 3.2).
- We give pseudocode for the log sealing algorithm in *journald* which is not secure in our model (Section 3.3).
- We propose a logging scheme and show that it satisfies our definition of security (Section 3.4) and prove it secure (Section 3.5). Since patches 1 and 2 from Section 5 have been merged, our logging scheme is the basis for the log sealing in *journald*.
- We discuss the individual issues and patches on a practical level (Section 5).

1.2 Related Work

Log sealing is a long standing subject of cryptographic research as one instance of forward security [1], yet it is still not prevalent in general purpose computer systems.

Forward secure log sealing schemes can be divided into two categories: symmetric authentication and asymmetric authentication (publicly verifiable).

Recent research on symmetric authentication focuses on the danger of storing, sending and sealing log messages asynchronously [2] (with improvements in [3]), as is common for most logging systems to not impact the application's performance. In contrast, preventing log tampering with events just before the compromise is definitely out of *journald*'s scope, as *journald* does not seal messages one at a time, but uses time-based epochs. *Journald* trades this benefit for allowing to verify log excerpts with epoch granularity. As the system by Paccagnella et al. does not bind the MAC keys to real time, the whole log from the beginning needs to be verified, otherwise an attacker could forge seemingly old entries and seal them with a new key. This line of research which constructs symmetric forward-secure logging schemes started with [4], and the algorithm in *journald* and our analysis of it builds upon this line.

Research for asymmetric authentication focuses on aggregate signatures. In order to amortize the size of the signatures, which are generally larger than the symmetric MAC tags, asymmetric schemes often propose to aggregate all signatures in a log file to compress them. Building on this, one interesting variation is the possibility to still verify excerpts. Such an approach was presented by Hartung [5], where the author formally models log schemes and provides security notions. The work of [5] and [6] also feature a complete security definition for a whole logging scheme, that includes truncation resistance, rather than a standalone definition for a primitive.

1.3 Outline

In Section 2 we introduce the necessary building blocks, namely seekable sequential key generators (SSKG) and forward secure message authentication schemes (FS-MAC). In Section 3 we introduce the notion of forward secure *seekable* MAC (FSS-MAC) and provide a construction based on a SSKG and FS-MAC. We then use the FSS-MAC to construct a forward secure logging scheme that is the basis for the patches that we present in Section 5. Furthermore we describe our security definition for logging schemes and show that our construction is secure according to our definition. In Section 4 we turn towards the practical issues and describe the *journald* file structure in detail to prepare for explaining the vulnerabilities and its patches in Section 5. Finally, in Section 6 we describe our toolkit for inspecting and modifying journal files that we use to showcase the attacks described in Section 5.

2 Preliminaries

The cryptographic foundation of log sealing in *journald* is the work on seekable sequential key generators by Marson and Poettering [7], which also details their application and implementation into *journald* for log sealing. Their construction describes a way to generate a forward-secure sequence of keys that additionally allows efficient seeking. Seeking allows computing the n-th key of the sequence without needing to calculate the intermediate keys. Those keys are then used for a message authentication code instantiated with HMAC [8] to seal sections of a log file. We start by introducing the notations and (security) definitions for a seekable sequential key generator `SSKG` and a forward-secure message authentication code `FS-MAC`.

Notation We indicate an empty list with $[\,]$. To append an entry e to a list list, we use $\mathit{list}\|e$. We assume that lists and intervals $[a, b]$ can be iterated over in sequence using the syntax "For $i \in \mathit{list}$".

2.1 Seekable Sequential Key Generator SSKG

Seekable sequential key generators (SSKG), introduced by Marson and Poettering [7], are based on sequential key generators, introduced in the same work. SSKG is a similar primitive as stateful generators, introduced by Bellare and Yee [4]. A SSKG is a mechanism or algorithm designed to generate a sequence of keys in a manner that allows efficient seeking or retrieval of specific keys within the sequence. A key generator is seekable if there is an additional algorithm Seek with the following syntax. The seekability is an efficiency feature instead of a security property. The SSKG has to be correct with respect to Seek.

Definition 1 (Seekable Sequential Key Generator SSKG [7]). *The interface of a SSKG is given by a set of four probabilistic polynomial time (PPT) algorithms and a deterministic algorithm GetKey:*

$$\begin{aligned}
\texttt{KeyGen}: &\quad (1^\lambda) \mapsto (pp, sek)\\
\texttt{StateGen}: &\quad (pp) \mapsto (st_0)\\
\texttt{Update}: &\quad (st_i) \mapsto (st_{i+1})\\
\texttt{Seek}: &\quad (sek, st_0, i) \mapsto (st_i)\\
\texttt{GetKey}: &\quad (st_i) \mapsto (k_i).
\end{aligned}$$

KeyGen generates on input of the security parameter λ some public parameters pp and a seeking key sek. An initial state st_0 is generated by StateGen on the given public parameters pp. The states are supposed to be kept secret and can be seen as mostly equivalent to secret keys. Updating one state to the next one via Update enables forward security if the old state is securely deleted after an update. The Seek algorithm gives one the possibility to efficiently calculate an arbitrary state st_i without having to calculate all intermediate states. The deterministic algorithm GetKey of SSKG outputs the corresponding k_i from a keyspace $\mathcal{K}$ to a state st_i on input of this state. We expect a SSKG to fulfill its notion of correctness, i.e. that whenever $(pp, sek) \leftarrow \texttt{KeyGen}(1^\lambda)$, and $st_0 \leftarrow \texttt{StateGen}(pp)$, then $\forall i \in \mathbb{N} : \texttt{Seek}(sek, st_0, i) = \texttt{Update}^i(st_0)$. This implies that Update and Seek are deterministic.

Definition 2 (FS-RoR-IND of SSKG, adapted from [7]). *A seekable sequential key generator SSKG is forward-secure real-or-random indistinguishable against adaptive adversaries (FS-RoR-IND) if for all PPT adversaries $\mathcal{A}$ that interact in experiment $\mathsf{Exp}^{\texttt{FS-RoR-IND},b}_{\texttt{SSKG}}$ from Figure 3, with $b = 0$ being the random and $b = 1$ the real game, the following advantage function is negligible:*

$$\mathsf{Adv}^{\texttt{FS-RoR-IND}}_{\texttt{SSKG},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathsf{Exp}^{\texttt{FS-RoR-IND},1}_{\texttt{SSKG},\mathcal{A}} = 1\right] - \Pr\left[\mathsf{Exp}^{\texttt{FS-RoR-INDFS},0}_{\texttt{SSKG},\mathcal{A}} = 1\right]\right|.$$

The goal of $\mathcal{A}$ in the FS-RoR-IND game of SSKG Figure 3 is to guess if the key k_n^b is real or random. It can query multiple times an oracle which returns a key k_i on query i. At some point $\mathcal{A}$ sends two indices n and m to challenger $\mathcal{C}$ to get the n-th key of the seekable sequential key generator k_n^b and the state st_m

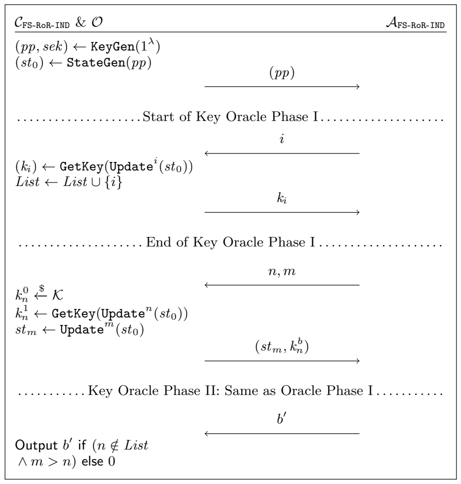
$\mathcal{C}_{\texttt{FS-RoR-IND}}$ & $\mathcal{O}$
$\mathcal{A}_{\texttt{FS-RoR-IND}}$
$(pp, sek) \leftarrow \texttt{KeyGen}(1^\lambda)$
$(st_0) \leftarrow \texttt{StateGen}(pp)$
(pp)
Start of Key Oracle Phase I
i
$(k_i) \leftarrow \texttt{GetKey}(\texttt{Update}^i(st_0))$
$List \leftarrow List \cup \{i\}$
k_i
End of Key Oracle Phase I
n, m
$k_n^0 \xleftarrow{\$} \mathcal{K}$
$k_n^1 \leftarrow \texttt{GetKey}(\texttt{Update}^n(st_0))$
$st_m \leftarrow \texttt{Update}^m(st_0)$
(st_m, k_n^b)
Key Oracle Phase II: Same as Oracle Phase I
b'
Output b' if $(n \notin List$
$\wedge\, m > n)$ else 0

Fig. 3. The `FS-RoR-IND` Game $\mathsf{Exp}_{\texttt{SSKG}}^{\texttt{FS-RoR-IND},b}$ for `SSKG`: The adversary $\mathcal{A}$ has to guess if a challenge key k_n^b by challenger $\mathcal{C}_{\texttt{FS-RoR-IND}}$ is real or random with the access to an oracle $\mathcal{O}$ for key determination.

for the m-th call of `Update`. To be sure that $\mathcal{A}$ can not determine the key of the m-th state naturally, m has to be larger than n. $\mathcal{A}$ can now call the same oracle again multiple times before it guesses if the challenged k_n^b is real or random.

We focus on the definitions by [7] as they constructed a `SSKG` from *factorization-based shortcut permutations* which yields in combination with a cryptographic MAC, see Appendix A, the implementation in *journald*.

2.2 Forward-Secure Message Authentication Scheme `FS-MAC`

Forward security in message authentication schemes refers to the property that a compromised key in the future cannot be used to retroactively forge MACs for messages that were logged in the past. Definitions for message authentication schemes `MAC` are provided in Appendix A. A MAC is generated for every message. After the end of one epoch the key is updated. In other words, even if an attacker gains access to the secret key used to generate MACs in this epoch, they cannot use that knowledge to create new MACs for modified past messages, i.e. forge MACs from a former epoch.

Definition 3 (Forward-Secure Message Authentication Scheme `FS-MAC` adapted from [4]). *The interface of a* `FS-MAC` *is given by a set of four PPT algorithms:*

$$\begin{aligned}
\texttt{KeyGen}: &\quad (1^\lambda) \mapsto (k_0)\\
\texttt{Update}: &\quad (k_i) \mapsto (k_{i+1})\\
\texttt{Sign}: &\quad (k_i, msg) \mapsto (tag)\\
\texttt{Vfy}: &\quad (k_i, msg, tag) \to \{0,1\}.
\end{aligned}$$

In addition to a standard `MAC` scheme a `FS-MAC` provides an `Update` function to update the key from one epoch to the next one. Hence, the keys k and the tags tag are indexed. We expect a `FS-MAC` to fulfill the notion of correctness, i.e. that whenever $(k_0) \leftarrow \texttt{KeyGen}(1^\lambda)$ and $\forall i > 0 : (k_{i+1}) \leftarrow \texttt{Update}(k_i)$, then $\forall i : (tag) \leftarrow \texttt{Sign}(k_i, msg)$ it holds that $\texttt{Vfy}(k_i, msg, tag) = 1$.

Extending the unforgeability notion (`EUF-CMA`) of a MAC scheme, we require that a `FS-MAC` is unforgeable in a forward-secure sense. Meaning that an adversary gets the additional power to update keys via an *Epoch Switching Oracle* in the first oracle phase. But more importantly, the adversary can break into the system and get the key of the current epoch. Hence it can obviously generate valid pairs of message msg and tag tag for the current and future epochs, but it should still not be possible that an adversary computes a correct pair of message and tag for past epochs.

Definition 4 (`FS-EUF-CMA` of `FS-MAC`). *A `FS-MAC` is forward-secure existentially unforgeable under chosen message attacks (`FS-EUF-CMA`) if for any PPT adversaries $\mathcal{A}_{\texttt{FS-EUF-CMA}}$ the advantage to win the `FS-EUF-CMA` game shown in Figure 4 is negligible in λ.*

The number of the current epoch, stated by counter i, increases with every key update. The counter j indicates the number of tags, i.e. the signatures of the `MAC`. Both counters are independent variables, e.g., the amount of tags per epoch can be greater than one. A `FS-MAC` can be constructed from a message authentication scheme (Definition 8) by using keys generated from an `SSKG`. In *journald* the `SSKG` is the one presented in [7] based on factoring and the `MAC` is HMAC (Hash-based Message Authentication Code) [8].

3 Security Analysis of a Forward-Secure Symmetric Log Scheme

We define in Section 3.1 a forward-secure seekable message authentication scheme `FSS-MAC` combining the security properties of a forward-secure message authentication scheme `FS-MAC` given in Section 2.2 with the seeking feature of a seekable sequential key generator `SSKG` given in Section 2.1. We construct the `FSS-MAC` from a `MAC`, i.e. HMAC, with keys provided by a `SSKG`.

Building on `FSS-MAC`, we define in Section 3.2 a forward-secure log scheme `LS` together with its `FS-EUF-CLMA` security definition. In this framework we give pseudocode corresponding to the original implementation of *journald* in Section 3.3 and give an intuition of its security vulnerabilities. We provide an improved version of the vulnerable log scheme (Section 3.4) which we prove secure (Section 3.5) and which is the basis for the patches that we provide in Section 5.

3.1 Forward-Secure Seekable Message Authentication

In the following definition of a forward-secure seekable message authentication scheme, we highlight in orange the modifications we made to include the function

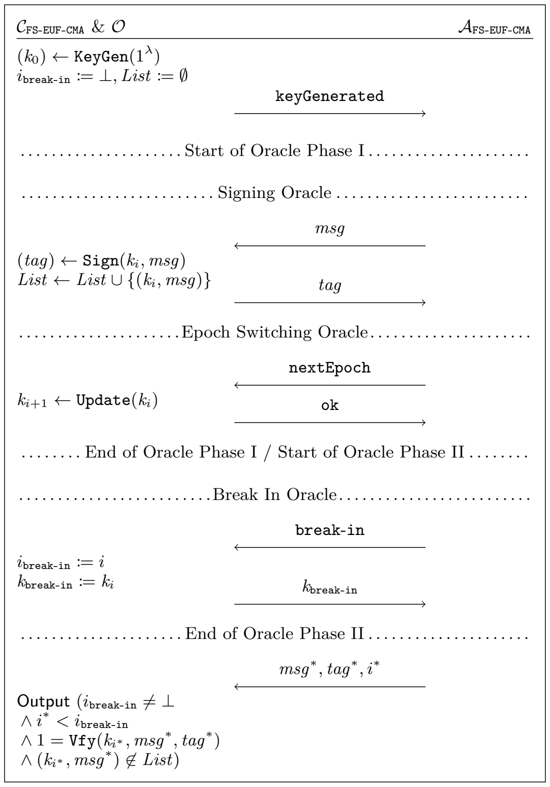
$\mathcal{C}_{\texttt{FS-EUF-CMA}}$ & $\mathcal{O}$
$\mathcal{A}_{\texttt{FS-EUF-CMA}}$
$(k_0) \leftarrow \texttt{KeyGen}(1^\lambda)$
$i_{\texttt{break-in}} := \perp, \mathit{List} := \emptyset$
keyGenerated
Start of Oracle Phase I
Signing Oracle
msg
$(\mathit{tag}) \leftarrow \texttt{Sign}(k_i, \mathit{msg})$
$\mathit{List} \leftarrow \mathit{List} \cup \{(k_i, \mathit{msg})\}$
tag
Epoch Switching Oracle
nextEpoch
$k_{i+1} \leftarrow \texttt{Update}(k_i)$
ok
End of Oracle Phase I / Start of Oracle Phase II
Break In Oracle
break-in
$i_{\texttt{break-in}} := i$
$k_{\texttt{break-in}} := k_i$
$k_{\texttt{break-in}}$
End of Oracle Phase II
$\mathit{msg}^*, \mathit{tag}^*, i^*$
Output $(i_{\texttt{break-in}} \neq \perp$
$\wedge\, i^* < i_{\texttt{break-in}}$
$\wedge\, 1 = \texttt{Vfy}(k_{i^*}, \mathit{msg}^*, \mathit{tag}^*)$
$\wedge\, (k_{i^*}, \mathit{msg}^*) \notin \mathit{List})$

Fig. 4. The FS-EUF-CMA Game for FS-MAC

of *seeking* in the definition of forward-secure message authentication of Definition 3. *Seeking* allows computing the current state/key from the initial state/key and the seeking key without the need to compute the intermediate states/keys. *Seeking* is a function to improve the efficiency of a scheme and does not affect its security. We include this feature to be closer to the real-world, i.e. the implementation of *journald*.

Definition 5 (Forward-Secure Seekable Message Authentication Scheme FSS-MAC). *The interface of a* FSS-MAC *is given by a set of five PPT algorithms:*

$$\begin{aligned}
\texttt{KeyGen}: &\quad (1^\lambda) \mapsto (\mathit{sek}, \mathit{st}_0)\\
\texttt{Update}: &\quad (\mathit{st}_i) \mapsto (\mathit{st}_{i+1})\\
\texttt{Sign}: &\quad (\mathit{st}_i, \mathit{msg}) \mapsto (\mathit{tag})\\
\texttt{Seek}: &\quad (\mathit{sek}, \mathit{st}_0, i) \mapsto (\mathit{st}_i)\\
\texttt{Vfy}: &\quad (\mathit{st}_i, \mathit{msg}, \mathit{tag}) \to \{0,1\}.
\end{aligned}$$

The algorithm `KeyGen` generates key *sek* for the seeking ability together with the first state/key st_0. The latter will be updated to the next state by the algorithm `Update`. The repetitive call of $\texttt{Update}^n$ results in the n-th state. Depending on its implementation, the `Seek` algorithm can be used to calculate directly the n-th state with the knowledge of the secret seeking key *sek*. Note that the `Seek` algorithm provides only efficiency and is not required to construct a forward-secure `MAC`. The `Sign` and `Vfy` algorithm of a `FSS-MAC` work as expected: `Sign` is a `MAC` on a message *msg* and state st_i outputting the related tag *tag*. On input of all last three variables, `Vfy` checks whether they fit together.
We expect a `FSS-MAC` to fulfill the following three properties: Notion of correctness, i.e. whenever $(sek, st_0) \leftarrow \texttt{KeyGen}(1^\lambda)$ and $\forall i \geq 0 : (st_{i+1}) \leftarrow \texttt{Update}(st_i)$, then $\forall i : st_i = \texttt{Seek}(sek, st_0, i)$ and $\forall i : (tag) \leftarrow \texttt{Sign}(st_i, msg)$ it holds that $\texttt{Vfy}(st_i, msg, tag) = 1$. Notion of efficiency, i.e. the time t' to seek a particular state is restricted to $t' = o(t(i))$, where $t(i)$ is the time taken to call $\texttt{Update}^i$. Notion of security, i.e. the scheme fullfils Definition 4 with the adjustment that the seeking key *sek* output by `KeyGen` is ignored. This is reasonable because, in the setting of Fig. 1, the verification system V receives as verification keys the first state/key st_0 and the seeking key *sek*, whereas the host system H only keeps the current state st_i.

Note that, as for `SSKG` the correctness notion implies `Update` and `Seek` to be deterministic.

Construction 1 (Forward-Secure Seekable Message Authentication Scheme `FSS-MAC`). Given a seekable sequential key generator `SSKG` as defined in Definition 1 and a message authentication scheme `MAC` (as defined in Definition 8 in the appendix), we construct a forward-secure seekable message authentication scheme `FSS-MAC` given in Figure 5.

```
KeyGen($1^\lambda$):
  – $(pp, sek) \leftarrow$ SSKG.KeyGen($1^\lambda$)
  – $(st_0) \leftarrow$ SSKG.StateGen($pp$)
↪ Output $(sek, st_0)$
Update($st_i$):
  – $(st_{i+1}) \leftarrow$ SSKG.Update($st_i$)
↪ Output $(st_{i+1})$
Sign($st_i, msg$):
  – $k_i \leftarrow$ MAC.KeyGen($1^\lambda$; SSKG.GetKey($st_i$))
  – $tag \leftarrow$ MAC.Sign($k_i, msg$)
↪ Output $(tag)$
Seek($sek, st_0, i$):
  – $st_i \leftarrow$ SSKG.Seek($sek, st_0, i$)
↪ Output $(st_i)$
Vfy($st_i, msg, tag$):
  – $k_i \leftarrow$ MAC.KeyGen($1^\lambda$; SSKG.GetKey($st_i$))
↪ Output MAC.Vfy($k_i, msg, tag$)
```

Fig. 5. The Construction of Forward-Secure Seekable Message Authentication Scheme `FSS-MAC`. Note that "`SSKG.GetKey`(st_i)" is a source of randomness for the MAC key generation algorithm.

Construction 1 in Figure 5 gives a generic construction of a `FSS-MAC`. In order to derive the MAC keys using `SSKG` we use the generated keys as randomness for

`MAC.KeyGen`. The security notion `FS-EUF-CMA` of `FS-MAC` (Definition 4) can directly be used as notion for `FSS-MAC` (the challenger simply discards the seeking key). The security of Construction 1 is the same as the security of the same scheme without `Seek` algorithm, as the additional key from `KeyGen` is not given to the adversary and the existence of another algorithm as such does not give the adversary any advantage.

Lemma 1. *If* `SSKG` *is a* `FS-RoR-IND` *secure seekable sequential key generator and* `MAC` *is a* `EUF-CMA` *secure MAC, then the* `FSS-MAC` *scheme in Figure 5 is* `FS-EUF-CMA` *secure.*

Proof (Proof sketch). Starting with the `FS-EUF-CMA` game, a security argument can be given via a series of hybrid games, where the intermediate keys are not generated by the `SSKG`, but are drawn by the challenger at random. The attacker cannot distinguish two adjacent hybrid games, because of the `FS-RoR-IND` security of `SSKG`. After the hybrid steps in each epoch a proper `MAC` key, i.e. generated with `KeyGen` from uniform randomness, is used. To reduce to the `EUF-CMA` security of `MAC`, the reduction guesses the epoch i^* for which $\mathcal{A}$ outputs a forgery. Then the reduction uses $\mathcal{C}_{\texttt{EUF-CMA}}$ to answer the signing queries of $\mathcal{A}$ for epoch i^*. If $\mathcal{A}$ outputs forgery for epoch i^*, the reduction uses this to win the `EUF-CMA` game.

3.2 Forward-Secure Seekable Log Scheme

A log scheme is supposed to be run by a log host, an auditor which verifies sealed logs and an application which sends log messages to the log host. Bases for our model are taken from the forward-secure logging models of [5, 6]. However, these definitions do not fit the log scheme of *journald* as they are using asymmetric signing keys and do not incorporate seeking, which is the primary feature of *journald*'s implementation.

We define a logging scheme `LS` which is based on a forward-secure seekable message authentication scheme as presented in Section 3.1. One main goal of the description of `LS` is to provide a better connection between theory and practice, as `LS` is the basis for the patches that we provide in Section 5.

Definition 6 (Logging scheme `LS`). *The interface of a log scheme* `LS` *is given by a set of five PPT algorithms:*

$$\begin{aligned}
\texttt{KeyGen}: &\quad (1^\lambda) \mapsto (sek, state)\\
\texttt{Sign}: &\quad (state) \mapsto (state', journal)\\
\texttt{AppendAndSeal}: &\quad (state, msg, time) \mapsto (state', journal)\\
\texttt{Vfy}: &\quad (sek, journal, target) \to \{0,1\}\\
\texttt{Query}: &\quad (journal, target) \mapsto (msg)
\end{aligned}$$

The key generation algorithm `KeyGen` generates a key *sek* for the fast-forward seeking ability together with the first state/key *state*. The *state* will be updated by the `Sign` and `AppendAndSeal` algorithm. Semantically, the difference between *signing* and *sealing* is that we require a sealing algorithm to evolve the key (if needed) and to securely erase the old key. It is important that the old key is *securely erased*, which is not easy to guarantee and is a research topic on its own [9]. A *journal* as output of the `Sign` and `AppendAndSeal` algorithm is the representation of the whole log including all log messages, tags, and counters. The `Sign` algorithm signs the current *journal*, but does not evolve the key. In the real *journald* application this happens when the log service is stopped (e.g. when the system is rebooted). From a security standpoint, it serves no functional purpose, as the old key remains available. The `AppendAndSeal` algorithm appends a message *msg*. The variable *time* represents the time of the log message. If the given time falls out of the current epoch, `AppendAndSeal` *seals* the log by updating the key and deleting the old one. The verifying algorithm `Vfy` checks whether the *journal* until epoch *target* is valid. This differs slightly from the interface for the verification in *journald*'s implementation, but as the main difference is a linear performance overhead we chose to model the simpler version. The algorithm `Query` will output the messages from the log *journal* from the first until a given target epoch *target*.

We expect a log scheme `LS` to fulfill the notion of correctness: After an arbitrary sequence of `Sign` and `AppendAndSeal` invocations, with the last message having time $time^*$, for all target epochs *target* with $\texttt{EpochStart}(target + 1) \leq time^*$: $\texttt{Vfy}(sek, journal, target) = 1$ and $\texttt{Query}(journal, target)$ returns all messages with $time < \texttt{EpochStart}(target + 1)$, logged with `AppendAndSeal`. $\texttt{EpochStart}(i)$ outputs the time *time* at the start of the i-th epoch.

We now define our security notion for a forward-secure log scheme based on symmetric cryptography and time-based epochs. Our security notion of *forward-secure existential unforgeability of chosen log-message attacks* `FS-EUF-CLMA` for symmetric log schemes is related to the security model of the secure logging schemes with verifiable excerpts in [5]. We show that the constructed log scheme `LS` of Section 3.4 satisfies the security notion of `FS-EUF-CLMA` for symmetric log schemes by a reduction to the `FS-EUF-CMA` security of the underlying `FSS-MAC` scheme.

Definition 7 (`FS-EUF-CLMA` of `LS`). *A log scheme* `LS` *is forward-secure existentially unforgeable under chosen log message attacks (*`FS-EUF-CLMA`*) if for any PPT adversary* $\mathcal{A}_{\texttt{FS-EUF-CLMA}}$*, that submits monotonically increasing time, the advantage to win the* `FS-EUF-CLMA` *game shown in Figure 6 is negligible in the security parameter* λ*.*

The `FS-EUF-CLMA` definition provides some more real-world aspects of a log scheme. In the security experiment we will let a challenger play the role of the log host and auditor, while the adversary can send messages with the goal to forge log messages in the end. At the beginning the challenger $\mathcal{C}_{\texttt{FS-EUF-CLMA}}$ as log server generates the secret key/state together with an empty *journal*. The challenger maintains the internal state of `LS` and the adversary can append messages and seal the journal via oracle access. We note that when an application sends `init`

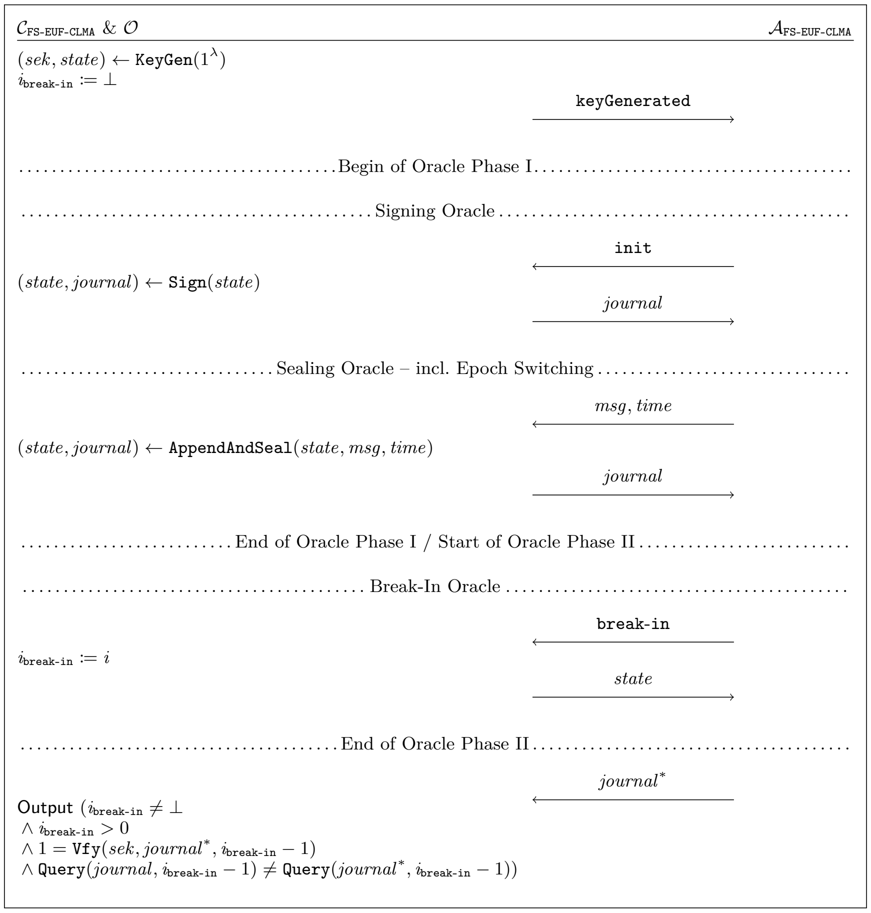
$\mathcal{C}_{\texttt{FS-EUF-CLMA}}$ & $\mathcal{O}$
$\mathcal{A}_{\texttt{FS-EUF-CLMA}}$
$(sek, state) \leftarrow \texttt{KeyGen}(1^\lambda)$
$i_{\texttt{break-in}} := \perp$
keyGenerated
Begin of Oracle Phase I
Signing Oracle
init
$(state, journal) \leftarrow \texttt{Sign}(state)$
journal
Sealing Oracle – incl. Epoch Switching
msg, time
$(state, journal) \leftarrow \texttt{AppendAndSeal}(state, msg, time)$
journal
End of Oracle Phase I / Start of Oracle Phase II
Break-In Oracle
break-in
$i_{\texttt{break-in}} := i$
state
End of Oracle Phase II
$journal^*$
Output $(i_{\texttt{break-in}} \neq \perp$
$\wedge\, i_{\texttt{break-in}} > 0$
$\wedge\, 1 = \texttt{Vfy}(sek, journal^*, i_{\texttt{break-in}} - 1)$
$\wedge\, \texttt{Query}(journal, i_{\texttt{break-in}} - 1) \neq \texttt{Query}(journal^*, i_{\texttt{break-in}} - 1))$

Fig. 6. The FS-EUF-CLMA security game for LS

to the host—triggered in practice e.g. after the suspension of a system—then the actual log will only be signed but not sealed. The sealing takes place, when a log message is processed: if the message to be appended falls outside the current epoch, the currently pending messages are sealed and afterwards the message is appended. In the security game the adversary $\mathcal{A}_{\texttt{FS-EUF-CLMA}}$ can provoke the signing of a current journal without key evolution by querying `init` to the Signing Oracle during the first oracle phase. During the same oracle phase $\mathcal{A}_{\texttt{FS-EUF-CLMA}}$ can append messages to *journal* via queries to the Sealing Oracle. Depending on the time in $(msg, time)$ this query provokes a sealing of the journal including an evolution of the epoch counter i. We note that the adversary can query both oracles arbitrarily often and in arbitrary order during oracle phase one. This phase ends with the break-in of the adversary. At this point, $\mathcal{A}_{\texttt{FS-EUF-CLMA}}$ receives the current state from the challenger who accepts no further queries apart from the potential forgery. In contrast to the FS-EUF-CMA notion of the

FSS-MAC the adversary gives a whole log $journal^*$ to forge the scheme instead of only a message and tag pair, but both are included in the log as well as the targeted epoch. We ask for $i_{\texttt{break-in}} > 0$ to have at least one journal to argue the security over it. If the journal, i.e. the state, is empty the adversary could submit a $journal^*$ which is obviously no forgery as it only contains content for messages during or after the epoch of break-in. It is trivial for an adversary to forge a log message after and during the epoch of the break-in as it gets the the complete internal state of LS and can run AppendAndSeal themselves. Also truncating the log to the last epoch before the break-in is trivial for an attacker. Therefore, the security notion provides forward-secure existential unforgeability until the last epoch before the break-in. This is modeled by requiring an attacker to provide journals that differ when querying them until the epoch $i_{\texttt{break-in}}$.

3.3 Vulnerable Log Scheme in Journald

The construction of the log scheme in *journald* keeps track of log messages in blocks. Such a block is called *log* consisting of tuples $(msg, time)$, where $time$ is the time when the log message msg is emitted, e.g. $log_j = [(msg_0, time_0), (msg_1, time_1), (msg_2, time_2), \dots]$. Those logs are authenticated with a tag tag and, together with specific counters, are appended to *journal*. One counter (i) is the epoch counter which identifies which signing key was used for the tag, so it indicates the epoch where this log was appended. The other counter (j) assigns the number of performed sign-operations, i.e. j is the tag counter. An example of a *journal* is

$$journal = [(log_0, tag_0, 0, 0), (log_1, tag_1, 0, 1), (log_2, tag_2, 1, 2), (log_3, tag_3, 2, 3), ...].$$

The algorithm $\texttt{EpochStart}(i)$ returns the actual fixed time when the epoch i started. In practice the mapping between real time and epoch is chosen when the key is generated. By default in journald 252, the epoch starts with value zero at the time of key generation and increments every 15 minutes.

Construction 2 (Log Scheme used in Journald). Figure 7 shows the modelling of the original log scheme that was implemented in journald version 252. For FSS-MAC it uses Construction 1 where the SSKG is instantiated with the construction from [7].

While more low-level details of the log format are discussed in Section 4, we want to give an intuition on how this log scheme was designed to work. The KeyGen algorithm initializes the state and creates an FSS-MAC key. This includes the initialization of *journal*, *log*, and the counters i and j. When a portion of the log is to be signed, the log scheme uses the FSS-MAC to authenticate the current, pending block of journal messages *log*. When a log message is to be appended for some time *time* that falls outside of the current epoch (highlighted in AppendAndSeal), the scheme signs all messages (with $\texttt{Sign}'$) in the current epoch, and then evolves the key forward. This ensures that an attacker breaking in after this event, cannot create a MAC for that elapsed epoch. AppendAndSeal

```
KeyGen($1^\lambda$):
  – $(sek', st_0) \leftarrow$ FSS-MAC.KeyGen($1^\lambda$)
  – Initialize $log, journal := [\,]$
  – Initialize $i, j := 0$
  – $state := (log, journal, i, j, st_i)$
↪ Output $((sek', st_0), state)$

Sign'($state$):
  – Parse $(log, journal, i, j, st_i) := state$
  – $tag \leftarrow$ FSS-MAC.Sign($st_i, (log, i, j)$)
  – $journal \leftarrow journal \| (log, tag, i, j)$
  – Empty $log \leftarrow [\,]$
  – $j \leftarrow j + 1$
  – $state \leftarrow (log, journal, i, j, st_i)$
↪ Output $(state, journal)$

Sign($state$):
↪ Output Sign'($state$)

AppendAndSeal($state, msg, time$):
  – Parse $(log, journal, i, j, st_i) := state$
  – If ($time \geq$ EpochStart($i + 1$)):
      • $((log, journal, i, j, st_i), \cdot)$
        $\leftarrow$ Sign'($(log, journal, i, j, st_i)$)
      • While ($time \geq$ EpochStart($i+1$))
          ∗ $st_{i+1} \leftarrow$ FSS-MAC.Update($st_i$)
          ∗ Erase securely old state $st_i$
          ∗ $i \leftarrow i + 1$
  – $log \leftarrow log \| (msg, time)$
  – $state \leftarrow (log, journal, i, j, st_i)$
↪ Output $(state, journal)$

Vfy($sek, journal, target$):
  – Parse $(sek', st_0) := sek$
  – $\forall i \in [0, target]$ :
    $st_i \leftarrow$ FSS-MAC.Seek($sek', st_0, i$)
  – If not $\forall i \in [0, target]$
    $\forall (log, tag, i, j) \in journal$ :
    FSS-MAC.Vfy($st_i, (log, i, j), tag$), output 0
  – If the sequence of all epoch counters $i$
    in $journal$ is not monotonic, output 0
  – If the sequence of all tag counters $j$ in
    $journal$ is not consecutive, output 0
  – For $i \in [0, target], (log, \cdot, i, \cdot) \in journal$ :
      • Let $(msg^*, time^*) \in log$ be the
        last message/time pair in $log$
      • If $time^* \geq$ EpochStart($i+1$), output 0
↪ Output 1

Query($journal, target$):
  – $result := [\,]$
  – For $(log, \cdot, i, j)$
    $\in journal, (msg, time) \in log$:
      • If $time \geq$ EpochStart($target + 1$), break
      • $result \leftarrow result \| (msg)$
↪ Output $result$
```

Fig. 7. Pseudocode for the vulnerable Log Scheme in journald. Removals, compared to our improved version (Figure 8), are highlighted.

outputs an updated state that an attacker cannot see without breaking in, and the current journal, that an attacker gets immediately. To check a log for consistency, `Vfy` checks that all present MACs are authentic, and the last log message in a given epoch does not belong in the next epoch. `Sign` signs the current log messages without evolving the key. This corresponds to the behavior of *journald* when it is gracefully restarted. `Query` outputs in sequence all log messages of the journal in all epochs from the beginning up to the *target* epoch.

The pseudocode in Figure 7 is a simplification of the logic actually implemented in journald. For example this pseudocode models the journal as a simple list of tuples, while a real journal is a complex data structure of pointers, deduplicating common log messages and structures providing fast access into that list (see also Section 4). Also modeling `Vfy` and `Query` as functions taking a target epoch is a simplification. In reality both output more complex messages and leave for the user to decide which log entries they assume to be unaltered. One interesting change is, that `Vfy` only verifies *tag*s until a given target epoch *target*, while the real implementation, not knowing *target* will verify all `TagObjects`. We simplify those details to keep the model understandable while still exhibiting vulnerabilities we found in the real implementation.

In particular, the check in `Vfy` is insufficient. This check does not prevent old log messages from being signed with a newer key. An attacker can take a journal, remove seals, adjust log messages as desired, and re-create a seal with the current log sealing key. We elaborate on this issue in Section 5.1. The logic in `AppendAndSeal` skips over empty epochs. While at first glance this might seem reasonable to skip over epochs without any log messages, it enables an adversary to truncate a log and resume logging, without being detected. We elaborate on this issue in Section 5.2. The third issue (Section 5.3) is not visible on this abstraction level. In reality, a journal is a more complex data structure then the list modeled here (see Section 4), and the verification algorithm additionally verifies some consistency of it. These consistency checks are incomplete. Displaying the log uses this data structure to output only a desired subset of log messages. An adversary can provide an inconsistent datastructure that the verification algorithm will accept and where displaying will show the wrong set of messages.

3.4 Fixed Log Scheme for journald

We now give an improved log scheme construction that we prove `FS-EUF-CLMA` secure.

Construction 3 (Forward-Secure Log Scheme). Given a forward-secure seekable message authentication scheme `FSS-MAC` as defined in Definition 5, we construct a forward-secure log scheme `LS` given in Figure 8. We keep the changes compared to Construction 2 minimal, so they can be implemented in practice more easily. To make spotting the differences to Figure 7 easier, we highlight additions in Figure 8 and removals in Figure 7.

To resolve the first issue (Section 5.1), we check that the time for all messages sealed with a particular key is inside the epoch for that key. For simplicity, we removed the old check, as it only checks the last message in an epoch. An attacker forging a journal can easily produce non-monotonic times within an epoch, so without any additional check for monotonic time within an epoch, this does not add any value. For the real implementation of the change the additional check can be left in.

To resolve second issue (Section 5.2), we add a flag *isfinal* to all authenticated blocks, to indicate, whether a block is the last block of an epoch, actually sealing it. When sealing epochs we ensure that we do not skip epochs, but add empty *log*-blocks for every epoch that would be missing. In `Vfy` we check that for every epoch in the requested interval, at least one block with $\mathit{isfinal} = 1$ exists.

```
$\texttt{KeyGen}(1^\lambda)$:
  – $(sek', st_0) \leftarrow \texttt{FSS-MAC.KeyGen}(1^\lambda)$
  – Initialize $log, journal := [\,]$
  – Initialize $i, j := 0$
  – $state := (log, journal, i, j, st_i)$
$\hookrightarrow$ Output $((sek', st_0), state)$

$\texttt{Sign}'(state, isfinal)$:
  – Parse $(log, journal, i, j, st_i) := state$
  – $tag \leftarrow \texttt{FSS-MAC.Sign}(st_i, (log, i, j, isfinal))$
  – $journal \leftarrow journal \| (log, tag, i, j, isfinal)$
  – Empty $log \leftarrow [\,]$
  – $j \leftarrow j + 1$
  – $state \leftarrow (log, journal, i, j, st_i)$
$\hookrightarrow$ Output $(state, journal)$

$\texttt{Sign}(state)$:
$\hookrightarrow$ Output $\texttt{Sign}'(state, 0)$

$\texttt{AppendAndSeal}(state, msg, time)$:
  – Parse $(log, journal, i, j, st_i) := state$
  – While $(time \geq \texttt{EpochStart}(i + 1))$:
      • $((log, journal, i, j, st_i), \cdot)$
        $\leftarrow \texttt{Sign}'((log, journal, i, j, st_i), 1)$
      • $st_{i+1} \leftarrow \texttt{FSS-MAC.Update}(st_i)$
      • Erase securely old state $st_i$
      • $i \leftarrow i + 1$
  – $log \leftarrow log \| (msg, time)$
  – $state \leftarrow (log, journal, i, j, st_i)$
$\hookrightarrow$ Output $(state, journal)$

$\texttt{Vfy}(sek, journal, target)$:
  – Parse $(sek', st_0) := sek$
  – $\forall i \in [0, target]$ :
    $st_i \leftarrow \texttt{FSS-MAC.Seek}(sek', st_0, i)$
  – If not $\forall i \in [0, target]$
    $\forall (log, tag, i, j, isfinal) \in journal$ :
    $\texttt{FSS-MAC.Vfy}(st_i, (log, i, j, isfinal), tag)$,
    output 0
  – If the sequence of all epoch counters $i$
    in $journal$ is not monotonic, output 0
  – If the sequence of all tag counters $j$ in
    $journal$ is not consecutive, output 0
  – If not $\forall i \in [0, target]$ :
    $(\cdot, \cdot, i, \cdot, 1) \in journal$, output 0
  – If not $\forall (log, \cdot, i, \cdot, \cdot) \in journal$
    $\forall (\cdot, time) \in log$ :
    $\texttt{EpochStart}(i) \leq time <$
    $\texttt{EpochStart}(i + 1)$, output 0
$\hookrightarrow$ Output 1

$\texttt{Query}(journal, target)$:
  – $result := [\,]$
  – For $(log, \cdot, i, j, \cdot)$
    $\in journal, (msg, time) \in log$:
      • If $time \geq \texttt{EpochStart}(target + 1)$,
        break
      • $result \leftarrow result \| (msg)$
$\hookrightarrow$ Output $result$
```

Fig. 8. The Construction of Forward-Secure Seekable Log Scheme $\texttt{LS}$. Additions, compared to the vulnerable version (Figure 7), are highlighted.

Checking for multiple blocks is not necessary, as the key is deleted the moment a block with $isfinal = 1$ is produced. So in order to produce another block for the same epoch, an adversary would have to abort $\texttt{AppendAndSeal}$ or break the authentication scheme.

3.5 Security Analysis of the log schemes

An adversary using the log scheme construction modelled in Figure 7 can forge a $journal^*$ without breaking the $\texttt{FS-EUF-CMA}$ security of the $\texttt{FSS-MAC}$. It can make use of the first (Section 5.1) or second (Section 5.2) issue by exlpoiting the missing *isfinal*-check to produce a $journal^*$ which succeeds the verify algorithm and still holds $\texttt{Query}(journal, target) = \texttt{Query}(journal^*, target)$. This is possible because there are important checks missing in the verify algorithm of the vulnerable log scheme construction. Further details are discussed in Section 5. Next we show that an adversary using the improved log scheme construction modelled in Figure 8 is forced to forge the underlying `FSS-MAC` to break the `FS-EUF-CLMA` security.

Theorem 1. *The log scheme* `LS` *in Construction 3 is forward-secure existentially unforgeable under chosen log message attacks (*`FS-EUF-CLMA` *secure), if the underlying forward-secure message authentication scheme* `FSS-MAC` *is* `FS-EUF-CMA`*-secure.*

Proof. Given an adversary $\mathcal{A}$ on the `FS-EUF-CLMA` security of the log scheme `LS`, we can derive an adversary $\mathcal{B}$ on the `FS-EUF-CMA` security of the `FSS-MAC` scheme. In the following, we provide the reduction $\mathcal{B}$ from `LS`'s `FS-EUF-CLMA` security to the `FS-EUF-CMA` security of its `FSS-MAC` scheme.

On a highlevel view, $\mathcal{B}$ runs the `FS-EUF-CLMA` game code and if a *tag* from the `FSS-MAC` or an epoch evolving is required, $\mathcal{B}$ prompts its `FS-EUF-CMA` challenger $\mathcal{C}$. After break-in, $\mathcal{B}$ receives a forged $journal^*$ from $\mathcal{A}$. By using checks from `Vfy` algorithm $\mathcal{B}$ extracts a `FS-EUF-CMA` forgery for $\mathcal{C}$. Now, let us zoom in.

Oracle Queries: In the beginning, $\mathcal{B}$ initializes a *log* and a *journal* along with an epoch counter i and a tag counter j. Then it prompts `keyGenerated` to $\mathcal{A}$.
To answer `init` queries to the signing oracle, $\mathcal{B}$ sets the *isfinal*-flag to 0 and queries $MSG = (log, i, j, isfinal)$ as message to the signing oracle of its `FS-EUF-CMA` challenger. $\mathcal{B}$ receives back *tag* and appends it along with *MSG* to *journal*, empties *log*, increases the tag counter j, and sends $journal \leftarrow journal \| (log, tag, i, j, isfinal)$ to $\mathcal{A}$.
To answer $(msg, time)$ queries to the sealing oracle, $\mathcal{B}$ sets the *isfinal*-flag to 1 if *time* is newer than the start of the current epoch and queries $MSG = (log, i, j, isfinal)$ to its challenger. $\mathcal{B}$ appends the responded *tag* along with *MSG* to $journal \leftarrow journal \| (log, tag, i, j, isfinal)$, empties *log*, and increases the tag counter j. To evolve the epoch and key, $\mathcal{B}$ queries `nextEpoch` to its challenger and increases the epoch counter i on respond `ok`. When $time < \texttt{EpochStart}(i+1)$, $\mathcal{B}$ appends $(msg, time)$ to *log* and sends the current *journal* to $\mathcal{A}$.
To answer the (`break-in`) query to the break-in oracle, $\mathcal{B}$ forwards (`break-in`) to its challenger. On return of the current signing key $k_{\texttt{break-in}}$, $\mathcal{B}$ sends $state \leftarrow (log, journal, i_{\texttt{break-in}} \leftarrow i, j, st_i \leftarrow k_{\texttt{break-in}})$ to $\mathcal{A}$.

At some point after the break-in, $\mathcal{B}$ receives a $journal^*$ from $\mathcal{A}$. A *journal* is a sequence of blocks of the form $(log, tag, i, j, isfinal)$ with *log* consisting of $(msg, time)$ blocks. Hence, we have that $journal^* := [..., (log^*, tag^*, i^*, j^*, isfinal^*), ...]$ with $log^* = [..., (msg^*, time^*), ...]$. If $journal^*$ is no valid forgery such that $\mathcal{A}$ does not win the `FS-EUF-CLMA` game, then $\mathcal{B}$ does not need to win the `FS-EUF-CMA` game. From now on, we assume that $journal^*$ is a successful forgery such that $\mathcal{A}$ wins the `FS-EUF-CLMA` game. Then it holds that

– $i_{\texttt{break-in}} \neq \perp$,
– $i_{\texttt{break-in}} > 0$,
– $1 = \texttt{Vfy}(\mathit{sek}, \mathit{journal}^*, i_{\texttt{break-in}} - 1)$, which implies among other things that the sequence of all tag counters j in $\mathit{journal}^*$ is consecutive,
– and $\texttt{Query}(\mathit{journal}, i_{\texttt{break-in}} - 1) \neq \texttt{Query}(\mathit{journal}^*, i_{\texttt{break-in}} - 1)$.

For an easier comparison, $\mathcal{B}$ removes all log- and journal-entries from $\mathit{journal}$ with $\mathit{time} \geq \texttt{EpochStart}(i_{\texttt{break-in}})$. Let this be called $\mathit{journal}' = [..., (\mathit{log}, \mathit{tag}, i_{\texttt{break-in}} - 1, j, \mathit{isfinal})]$. This is valid as $\texttt{Query}(\mathit{journal}, i_{\texttt{break-in}} - 1)$ consists of all messages msg with $\mathit{time} < \texttt{EpochStart}(i_{\texttt{break-in}})$ which equals to all messages of log until epoch $i_{\texttt{break-in}}$. It follows that $\texttt{Query}(\mathit{journal}, i_{\texttt{break-in}} - 1)$ equals $\texttt{Query}(\mathit{journal}', i_{\texttt{break-in}} - 1)$.

As $\mathit{journal}^*$ is a successful forgery, it differs from $\mathit{journal}'$. Otherwise, if $\mathit{journal}^*$ equals $\mathit{journal}'$, then the $\texttt{Query}$ outputs are equal and $\mathcal{A}$ does not win the $\texttt{FS-EUF-CLMA}$ game.

$\texttt{CMA}$ *Forgery:* To extract a successful $\texttt{FS-EUF-CMA}$ forgery $(\mathit{MSG}^*, \mathit{tag}^*, i^*)$ from $\mathit{journal}^*$ with $\mathit{MSG}^* = \mathit{log}^*, i^*, j^*, \mathit{isfinal}^*$ $\mathcal{B}$ parses $\mathit{journal}^*$ and $\mathit{journal}'$ via the consecutive tag counters to get a list of all $\mathit{MSG} = (\mathit{log}, i, j, \mathit{isfinal})$. There exist three cases for $\mathit{journal}^*$ to differ from $\mathit{journal}'$:

1. We assume that $\mathit{journal}^*$ is a truncation of $\mathit{journal}'$, i.e. $\mathit{journal}^*$ is a prefix of $\mathit{journal}'$, i.e. the list of all MSG^* from $\mathit{journal}^*$ indexed by the tag counter is a prefix of the list of all MSG' from $\mathit{journal}'$. This means that either all blocks of the last epoch $i_{\texttt{break-in}} - 1$ are missing or at least the last entry with $\mathit{isfinal} = 1$ is missing. But in both cases $\texttt{Vfy}(\mathit{sek}, \mathit{journal}^*, i_{\texttt{break-in}} - 1)$ is not true as it verifies that for all epochs until $i_{\texttt{break-in}} - 1$ there is an entry $(\cdot, \cdot, i, \cdot, 1) \in \mathit{journal}^*$. Hence, $\mathit{journal}^*$ cannot be a truncation of $\mathit{journal}'$.
2. $\mathit{journal}^*$ is an extension of $\mathit{journal}'$, i.e. $\mathit{journal}'$ is a prefix of $\mathit{journal}^*$, i.e. the list of all MSG' from $\mathit{journal}'$ indexed by the tag counter is a prefix of the list of all MSG^* from $\mathit{journal}^*$.
If for all additional journal entries in $\mathit{journal}^*$: $i \geq i_{\texttt{break-in}}$, then all additional message have $\mathit{time} \geq \texttt{EpochStart}(i_{\texttt{break-in}})$ (as checked by $\texttt{Vfy}$), then $\texttt{Query}(\mathit{journal}^*, i_{\texttt{break-in}} - 1) = \texttt{Query}(\mathit{journal}', i_{\texttt{break-in}} - 1)$, because $\mathit{journal}^*$ and $\mathit{journal}'$ are identical on epochs $i < i_{\texttt{break-in}}$.
Otherwise, one of the additional journal entries $(\mathit{log}^*, \mathit{tag}^*, i^*, j^*, \mathit{isfinal}^*)$ in $\mathit{journal}^*$ has $i^* < i_{\texttt{break-in}}$. Furthermore, $\texttt{FSS-MAC.Vfy}(k_i^*, (\mathit{log}^*, i^*, j^*, \mathit{isfinal}^*), \mathit{tag}^*) = 1$, where k_i^* is the respective key of $\mathcal{C}$. (Otherwise $\mathit{journal}^*$ would not verify.) Finally, as the j's are consecutive, j^* is different to any j in $\mathit{journal}'$, whereby $\mathit{MSG}^* = (\mathit{log}^*, i^*, j^*, \mathit{isfinal}^*)$ has not been queried to $\mathcal{C}$ and is thus a successful forgery for the $\texttt{FS-EUF-CMA}$ game.
3. There exists a block $\mathit{MSG}^* = (\mathit{log}^*, i^*, j^*, \mathit{isfinal}^*)$ in $\mathit{journal}^*$ which differs from $\mathit{MSG}' = (\mathit{log}', i', j', \mathit{isfinal}')$ with $j^* = j'$, i.e. $\mathit{journal}^*$ and $\mathit{journal}$ have different sealed journal entries with the same tag counter.
If $\mathit{isfinal}' = 1$, MSG' is not in $\mathit{journal}^*$. In order for $\texttt{LS.Vfy}$ to acccept this log, there needs to be a (different) block $(\cdot, i', \cdot, 1) \in \mathit{journal}^*$. We use this

block as MSG^*. If $isfinal' = 0$, then $i' \geq i^*$, because as the epoch counters are monotonic, $i' < i^*$ would imply that a block $(\cdot, i', \cdot, 1)$ is missing in *journal** which would cause `LS.Vfy` to reject.
$\mathcal{B}$ submits (MSG^*, tag^*, i^*) to its challenger. As $\texttt{LS.Vfy}(sek, journal^*, i_{\texttt{break-in}} - 1) = 1$ it holds that $\texttt{FSS-MAC.Vfy}(k_i^*, MSG^*, tag^*) = 1$, where k_i^* is the respective key of $\mathcal{C}$. It holds $i^* \leq i' < i_{\texttt{break-in}}$, so the forgery is for an epoch i^* before break-in. As the sequence of all tag counters j in *journal** is consecutive, every MSG^* is unique which implies that MSG^* could not have been queried to the signing oracle of the `FS-EUF-CMA` game before.

Therefore, $\mathcal{B}$ wins the `FS-EUF-CMA` game. This is a contradiction to the assumption that the `FSS-MAC` in `LS` is `FS-EUF-CMA` secure. Hence, `LS` is proven to be `FS-EUF-CLMA` secure.

We conclude the following: Given a `FS-RoR-IND` secure seekable sequential key generator `SSKG` and a `EUF-CMA` secure message authentication scheme `MAC`, we can build a `FS-EUF-CLMA` secure log scheme. For this we first construct a forward-secure seekable message authentication scheme `FSS-MAC` from `SSKG` and `MAC` as shown in Figure 5. Then we use `FSS-MAC` to construct the log scheme as shown in Figure 8. The `FS-EUF-CMA` security of `FSS-MAC` follows from Lemma 1 and the `FS-EUF-CLMA` security of the log scheme follows from Theorem 1. This log scheme serves as basis for our patches of *journald* that we describe in Section 5.

4 The Journal File Format

While the formal model describes the journal file as a sequence of tuples, the reality looks more complex. In order to later understand how the practical attacks against the concrete implementation work, we will now take a closer look at how the file format looks in practice. Log messages are not plain strings but sets of key-value pairs, that store meta information in addition to the regular log messages. To allow quick access to sub-sequences of the journal (also based on the meta fields) and to not store multiple occurrences of the same value multiple times, journals are stored as a structure of linked objects indexed by hash tables.

4.1 Structure

For *journald* each log entry is, apart from time and sequence information, a set of key-value pairs encoding the log message, and other structured information like log-level, name of the logging program and systemd unit name. *Journald* stores logs in a binary file format on-disk. The individual structures are defined in `journal-def.h` [4], with a high-level description of the structure given in an accompanying documentation. [5]

[4] https://github.com/systemd/systemd/blob/v252/src/libsystemd/sd-journal/journal-def.h
[5] https://systemd.io/JOURNAL_FILE_FORMAT/

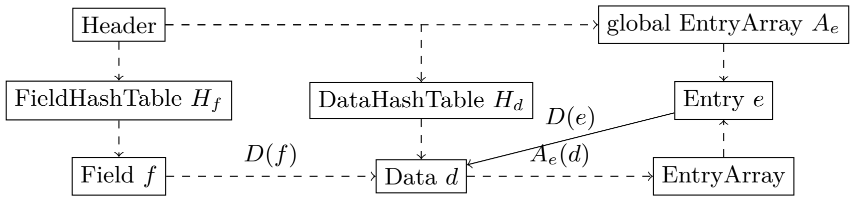

Fig. 9. Journal File Format, the `Object` suffix of each of the entity names (`EntryObject`, `FieldObject`, `DataObject`, `EntryArrayObject`) is omitted. Only the solid link between Entry and Data is authenticated by MACs

On a high level a journal file consists, after a header, of a sequence of objects encoded with type, length and payload. These objects contain data and references to each other as addresses (offsets from the start of the journal file), visualized in Figure 9. Each log entry is stored as an `EntryObject` e referencing multiple `DataObjects` $D(e)$, one for each key-value pair, reusing `DataObjects` that contain the same data. `FieldObjects` are used to link all `DataObjects` with the same key together.

For writing the journal file, the main design goal was to allow appending new log entries with only changing little of the existing journal file. Also highly repetitive messages (as well as repetitive values for other fields) should only be stored once. For reading, the file format allows to quickly seek to specific time ranges, display log messages with a specific value for a field, and listing all fields used and possible values for all those fields. These different access modes are supported by a collection of quick-access structures in the file:

– A hash table allowing to access `FieldObject` by value (H_f).
– A hash table allowing to access `DataObject` by value (H_d).
– The data objects chained in a linked list $D(f)$, allowing to list all individual values for field f.
– A global `EntryArray` A_e to list all entries in the log file.
– A chain of `EntryArrays` $A_e(d)$ per data object d, listing all usages of the data object.

Accessing the log file can happen in various ways. When a user queries for all log entries in a given range using the `journalctl` tool, that program will use the global `EntryArray` to iterate through the log entries and print the messages. When a user queries log entries for a given unit[6], `journalctl -u` will locate the corresponding `DataObject` and use the corresponding `EntryArrays` to print only the matching entries. When specifying the unit with a glob expression[7], instead of a concrete string, `journalctl` will use the unique values of that field to evaluate the glob expression on and then query log entries for all matching unique values. Special options of `journalctl` can be used to list all fields and all unique values of a field.

[6] systemd term for a service
[7] i.e. a pattern that allows wildcard expressions

4.2 Sealing with the Journal File Format

In order to seal the journal, `TagObject`s are regularly appended to the log file. They are numbered with an `epoch` and `seqnum`, corresponding to i and j from the theoretical model, respectively. They contain a MAC across parts of all objects until that point, starting from the last `TagObject` or the beginning of the file, for the first tag. The MAC covers the type and length for each object and all parts that are considered immutable (i. e. will not be modified when appending), in particular the actual data. This ensures that for all immutable parts the MAC secures the object's location in the journal file. To put it another way, without having a MAC key, the only messages an attacker can get accepted by the MAC-check are prefixes of the original log file where "mac-covered" regions are unchanged, and all other regions can contain arbitrary data. In particular, `FieldObject` and `DataObject` have their payload secured and `EntryObject`s have all pointers to `DataObject` secured. All structures and fields that are modified after writing the object initially are not covered by the MAC and their integrity relies on additional checks while verifying the journal.

5 Specific problems

We now describe the three individual issues that each allow different kind of forgeries and thereby break the security definition.

5.1 Sealing older entries with newer keys (CVE-2023-31439)

While the verification algorithm checks that the tags for a given epoch are not used to seal newer entries, the check in the other direction is missing allowing a key to seal arbitrary old journal entries. This allows an attacker to forge arbitrary old log entries and seal them with the new key, breaking all security guarantees that log sealing is trying to make. In particular an attacker having broken into a system, can use the current (newer) sealing key to generate new MACs for any previous `TagObject`s after tampering with the log entries. Verification would report that the sealing happened correctly and the modifications go unnoticed. If messages prior to the currently last tag should be added, that tag needs to be removed first.

This translates to a formal `FS-EUF-CLMA` attacker, who takes the last journal, uses the key from *state* from the Break-In Oracle, appends some log entries at the end, and uses the new key from *state* to re-seal the log. The attacker uses values of *time* before $\texttt{EpochStart}(i_{\texttt{break-in}})$. `Vfy` as implemented will not complain about the log messages sealed with a key not correspoding to the epoch they are in, and `Query` will output them. The attacker wins the security game, as `Query` with the modified journal will output the additional messages.

Remediation Recommendation We recommend to extend the check for log entries against crypto epochs to also report too old entries sealed with a given crypto key and not only too new entries. The journal writer as-is will not seal too old entries with a newer key, so this change should not report any false-positive violations. This patch was merged[8] in systemd version 255.

5.2 Attack on Completeness (CVE-2023-31438)

While log sealing can confirm the existence of log entries, it can also confirm the completeness of a given log section. In particular, if verification of a journal file succeeds for a given time frame the attacker should not be able to add, modify or remove any log entry in that time frame. This vulnerability allows an attacker under specific circumstances to truncate some log entries at the end of a log.

Journald's handling of epochs without log entries does not allow to verify the absence of log entries. Specifically an attacker breaking into a system can truncate previous sealed epochs of the log without being revealed when checking the log seals, as the journal is not updated when an epoch passes without new log entries; only the sealing key is evolved.

This translates to a formal `FS-EUF-CLMA` attacker, who creates a journal with two seals in the epoch before the break-in. From that journal the attacker removes the last seal (and corresponding message). `Vfy` as implemented will not complain (as there is still a seal for the last epoch) and the attacker wins the security game, as the truncated output differs from the original one.

Remediation Recommendation We recommend to create new `TagObject`s regardless of whether there are new log entries in the corresponding epoch, when moving to the next epoch. The `TagObject`s are relatively small in comparison to regular log entries and there are rarely epochs without log entries in practice. In the special case, when a system has been suspended for more than one epoch, *journald* would have to create `TagObject`s for every epoch that was missed. This can be made more efficient, by adding another field to the `TagObject`, indicating how many epochs are about to be skipped (and then be sealed with the old key). That way the journal only grows by a constant amount, and the missing empty epochs are still sealed. The verification process needs to be adjusted to recognize this new field (ensuring that each epoch is accounted for).

Partial epochs should not be sealed. As the sealing key is not rotated, there is no security benefit, This means, that un-sealed entries should be left as-is when *journald* is stopped. When logging resumes, the entries can be sealed if the epoch has ended, or get new entries appended, when it has not. Verification should then check that the epochs are strictly monotonic increasing.

A weaker variant of this change was merged[9]. For now, the additional seal when stopping *journald* was left in and the epoch-check was only tightened to check for continuously monotonic epochs and not for strictly monotonic epochs. This means, that if *journald* had been stopped in the last finished epoch where the break-in happens, attacker could truncated the messages after *journald* was

[8] https://github.com/systemd/systemd/pull/28885

[9] https://github.com/systemd/systemd/pull/28886

stopped and resume logging without being noticed, but arbitrary truncation is noticed.

5.3 Missing verifications of fast access structures (CVE-2023-31437)

This vulnerability allows an attacker to modify the set of displayed log entries, when a log is viewed with filters (like for example, displaying only log messages for a specific system service). The attack originates from missing consistency checks for the fast access structures, that allow filtering entries efficiently. We provide a patch that inserts them as `CVE-2023-31437.patch` in the repository for Section 6. For a more detailed description about the existing and missing checks, see the expanded version[10].

6 Implementation

For this work we created a small toolkit[11] allowing the close inspection and modification of existing journal files to demonstrate the described security flaws. In contrast, the implementation from systemd focuses on generating valid journal files and is optimized for appending to an existing journal file. Rewriting a journal file and adjusting pointers is not that easy with the library provided by systemd. However, the journal file that `journalctl -verify` consumes comes from a potentially compromised system, making it adversary-controlled input. In particular this means, that `journalctl -verify` needs to cope with any possible journal file contents. Our implementation allows the creation of unexpected journal files to simulate attacker behavior.

We provide an implementation of the `SSKG` construction by [7] in `FSPRNGKey` to be able to derive future sealing keys from the `fss`-file on disk. Additionally, we implement logic to parse the different journal objects in `JournalObject` to allow inspection and modification. That foundation is then used in `Attacker` to provide a concrete attacker against the security definition, demonstrating the issue from Section 5.1. To show that this attacker wins against the security definition we implement a specialization of the security experiment (in `Challenger.experimentWithJournal()`): We omit allowing the attacker to specify the log displaying query, as filtering the query for valid `journalctl` options is complex and not needed for this attack.

In addition we implement a test harness to allow the isolated invocation of a specific `journald` binary. The harness facilitates interaction between the security experiment and the journal process: One interaction is feeding specific log messages from the security experiment or simulated adversary, while regular log messages of the hosting system are not affected. Another interaction is to advance the epoch to force sealing of the log without having to wait for the epoch to expire normally. To advance epochs the harness uses libfaketime[12] and adjusts

[10] https://eprint.iacr.org/2023/867
[11] https://github.com/kastel-security/Journald
[12] https://github.com/wolfcw/libfaketime

the perceived system time for the `journald` process accordingly. Lastly, the test harness invokes `journald` using Bubblewrap (`bwrap`[13]) to link test directories to their expected locations. This is required, because `journald` searches for configuration, journal and the key files in hard coded paths. Bubblewrap allows to set up a mount namespace for the process under test, and adjust the directories visible under those paths. This is even allowed for non-privileged users, which is comfortable for running test cases and attacker simulations without requiring root privileges.

7 Disclosure

On 2023-01-18 we reached out to the security contact for systemd with a preliminary version of this article detailing the findings together with a high-level overview. On 2023-02-07 we got an acknowledgment, that the findings are being investigated. After some additional rounds of discussion, they stopped responding on 2023-03-20, without anything being acknowledged as vulnerability and no further action taken. On 2023-06-07 we published a version of the article via eprint and published the artifacts independently. At this point the CVEs were updated and marked as "disputed". On 2023-08-18 we submitted pull requests with the main proposed changes. On 2023-10-06 the changes for CVE-2023-31439 were merged. On 2023-11-08 the changes for CVE-2023-31438 were merged. On 2023-12-06 systemd v255 was released, which includes the changes. The CVE entries stay marked as "disputed" and inquiries from us, regarding the rationale stay unanswered.

8 Conclusion

We argue the security model presented by us in this paper is what a user relying on log sealing will reasonably expect. The changes proposed by us to resolve the major two of the vulnerabilities were merged from which we conclude that the authors at least deem the changes beneficial to their software.

Truncation resistance might be an often overlooked aspect of integrity but we believe it should be part of the security guarantee and it is included in our definition. Log sealing should detect truncation, also because it is not too difficult to achieve compared to the security benefit it brings, considering the end of the log file is usually the most important part after a breach. From a practical point of view log completeness should be further investigated for logs spanning multiple journal files.

In summary this work shows how large the gap between a security model and its implementation sometimes is, resulting in serious vulnerabilities. We improve this, by bringing more real-world complexity into the theoretical model and by resolving practical vulnerabilities in the implementation. Further improvements could be to incorporate even more real-world artifacts in the model, or adjust the implementation to have less quirks that deviate from the model.

[13] https://github.com/containers/bubblewrap

Acknowledgments. We thank the ACNS 2025 anonymous reviewers for their valuable feedback, and especially our Shepherd for the constructive and helpful discussion. We thank Alexander Koch for proofreading and Johannes Ottenhues for proofreading, discussions and helpful suggestions.
This work was supported by funding by the German Federal Ministry of Education and Research (BMBF) under the project "Sec4IoMT" (ID 16KIS1692) and the project "ASCOT", and by KASTEL Security Research Labs.

References

1. Bellare, M., and Yee, B.: Forward integrity for secure audit logs. Tech. rep., Citeseer (1997)
2. Paccagnella, R., Liao, K., Tian, D., and Bates, A.: Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1551–1574 (2020). DOI: 10.1145/3372297.3417862
3. Hoang, V.T., Wu, C., and Yuan, X.: Faster Yet Safer: Logging System Via Fixed-Key Blockcipher. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 2389–2406 (2022)
4. Bellare, M., and Yee, B.: Forward-Security in Private-Key Cryptography. In: Joye, M. (ed.) Topics in Cryptology — CT-RSA 2003, pp. 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). DOI: 10.1007/3-540-36563-X_1
5. Hartung, G.: Secure Audit Logs with Verifiable Excerpts. In: Sako, K. (ed.) Topics in Cryptology - CT-RSA 2016, pp. 183–199. Springer International Publishing, Cham (2016). DOI: 10.1007/978-3-319-29485-8_11
6. Hartung, G., Kaidel, B., Koch, A., Koch, J., and Hartmann, D.: Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures. In: Okamoto, T., Yu, Y., Au, M.H., and Li, Y. (eds.) Provable Security, pp. 87–106. Springer International Publishing, Cham (2017). DOI: 10.1007/978-3-319-68637-0_6
7. Marson, G.A., and Poettering, B.: Practical secure logging: Seekable sequential key generators. In: Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18, pp. 111–128 (2013). DOI: 10.1007/978-3-642-40203-6_7
8. Krawczyk, H., Bellare, M., and Canetti, R.: *HMAC: Keyed-Hashing for Message Authentication.* Request for Comments 2104. RFC Editor, Feb. 1997. 11 pp. DOI: 10.17487/RFC2104. https://www.rfc-editor.org/info/rfc2104.
9. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the Sixth USENIX Security Symposium, San Jose, CA, pp. 77–89 (1996)

A Message Authentication Code

A message authentication code—the common name for message authentication schemes—is a symmetrical way to authenticate a message by adding a *tag* via a keyed signing algorithm to any message. Message authentication schemes with symmetric cryptography are used to provide integrity protection and authenticity to messages. These schemes use a private key, which is kept secret, to generate a message authentication code (MAC) *tag* that can be used to verify the integrity of the message and authenticate the sender. Typically, a keyed hash-function is applied as signing algorithm, see, e.g. HMAC [8].

Definition 8 (Message Authentication Code MAC, based on [4]). *The interface of a MAC is given by a set of three PPT algorithms:*

$$\begin{aligned} \texttt{KeyGen}: &\quad (1^\lambda) \mapsto (k) \\ \texttt{Sign}: &\quad (k, msg) \mapsto (tag) \\ \texttt{Vfy}: &\quad (k, msg, tag) \rightarrow \{0,1\}. \end{aligned}$$

`KeyGen` generates a key k on input of the security parameter λ. With this key k can a sender generate a tag *tag* for any message *msg* by using the keyed signing algorithm `Sign`, which outputs for a message *msg* a tag *tag*. A verifier knowing the key k, the message *msg*, and the regarding tag *tag* can check the authenticity and integrity via the verifying algorithm `Vfy`, which returns either accept 1 or reject 0 on the input k, *msg*, and *tag*. We expect a MAC to fulfill the notion of correctness, i.e. that whenever $(k) \leftarrow \texttt{KeyGen}(1^\lambda)$, then $\texttt{Vfy}(k, msg, \texttt{Sign}(k, msg)) = 1$ for all messages *msg* and λ.

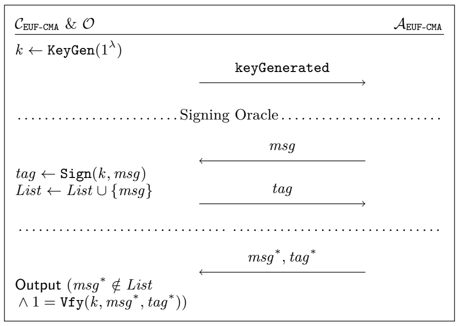
$\mathcal{C}_{\texttt{EUF-CMA}}$ & $\mathcal{O}$
$\mathcal{A}_{\texttt{EUF-CMA}}$
$k \leftarrow \texttt{KeyGen}(1^\lambda)$
keyGenerated
Signing Oracle
msg
$tag \leftarrow \texttt{Sign}(k, msg)$
$List \leftarrow List \cup \{msg\}$
tag
msg^*, tag^*
Output $(msg^* \notin List$
$\wedge\, 1 = \texttt{Vfy}(k, msg^*, tag^*))$

Fig. 10. The `EUF-CMA` Game for `MAC`

Definition 9 (`EUF-CMA` of `MAC`). *A* `MAC` *is existentially unforgeable under chosen message attacks (*`EUF-CMA`*) if for any PPT adversaries* $\mathcal{A}_{\texttt{EUF-CMA}}$ *the advantage to win the* `EUF-CMA` *game shown in Figure 10 is negligible in* λ.

The `EUF-CMA` security of `MAC` is defined over the game shown in Fig. 10 between a challenger $\mathcal{C}_{\texttt{EUF-CMA}}$ and an adversary $\mathcal{A}$ which can query a signing oracle $\mathcal{O}$ multiple times after receiving the prompt that the key was generated. $\mathcal{O}$ generates and outputs a tag *tag* for any message *msg* received by $\mathcal{A}$. In an empty initialized list *List* stores $\mathcal{O}$ the tuple of all queried *msg*. The goal of the game is that $\mathcal{A}$ tries to find a tuple of a message msg^* not queried to $\mathcal{O}$ before and tag tag^* such that the verify algorithm `Vfy` accepts the tuple.