Secure Logging with Crash Tolerance

Erik-Oliver Blass[1] and Guevara Noubir[2]

[1] Airbus Group Innovations, Munich, Germany
erik-oliver.blass@airbus.com
[2] Northeastern University, Boston, MA, USA
noubir@ccs.neu.edu

Abstract. Forward-secure logging protects old log entries in a log file against an adversary compromising the log device. However, we show that previous work on forward-secure logging is prone to *crash-attacks* where the adversary removes log entries and then crashes the log device. As the state of the log after a crash-attack is indistinguishable from the state after a real crash, e.g., power failure, the adversary can hide attack traces. We present SLiC, a new logging protocol that achieves forward-security against crash-attacks. Our main idea is to decouple the time of a log event with the position of its resulting log entry in the log file. Each event is encrypted and written to a pseudo-random position in the log file. Consequently, the adversary can only remove random log events, but not specific ones. Yet, during forensic analysis, the verifier can replay pseudo-random positions. This allows to distinguish a real crash (last events missing) from a crash-attack (random events missing). Besides a formal analysis, we also present an evaluation of SLiC as a syslog server to indicate its practicality.

1 Introduction

Virtually all modern computer systems, cell phones, and even embedded devices use logging services to store important information in a log file. Prominent examples for logging services in the Unix world are syslogd or journald. Log files are often used for system audits and during forensic analysis to learn about malfunction, attacks, and to detect system compromise.

However, in case an adversary fully compromises the device running the logging service, they can access and modify any information on that device. Typically, “covert” adversaries [1] then modify logs to hide traces of their attacks and remain undetected. If the verifier of a log file does not know whether entries in the log file have been tampered with, the log itself becomes useless for (forensic) analysis.

Consequently, a significant amount of work has investigated log integrity in case of log device compromise. Relying on hardware and trusted computing is often expensive and difficult to deploy on legacy devices, so research has mostly focused on software solutions. Their rationale is to provide *forward security* or *forward integrity* for log file entries [2, 4, 6, 10–14]. The main insight is that there cannot be security for log entries starting from the time of compromise t. By compromising the log device, the adversary has access to all information, even cryptographic secrets, and can fabricate log entries from then on. Yet, forward security ensures that all log entries from before time t are integrity protected. That is, if the adversary tampers with entries logged before time t, this will be detected by the verifier. This weak form of security (tamper evidence) is especially helpful in scenarios, where log entries before the time of compromise reveal an ongoing attack and will be integrity protected.

However in practice, log devices crash. Besides operating system crashes, log devices might operate in harsh environments and, e.g., experience power failure. Without strong assumptions on the operating system, file system, and cache, a crash can leave the log file in an inconsistent state. In the specific context of logging, this turns out to be a security challenge. In what we call a *crash-attack*, the adversary compromises the log device, modifies or removes log entries and integrity tags, and then purposefully crashes the log device. During forensic analysis, it is then difficult for a verifier to decide whether a crash and the inconsistent state and integrity information of the log is due to a (regular) crash or a crash-attack. None of the state of the art protocols for secure logging has been designed to cope with crashes, and their application in many real world scenarios is therefore risky.

A second challenge stems from the fact that either logging devices are resource-constrained, e.g., micro-controller based, or the amount of log entries per unit of time can become large. This requires secure logging protocols to be very lightweight.

In this paper, we design a new, efficient protocol for forward-secure logging that is "crash tolerant" and enables the verifier to distinguish a regular crash from a crash-attack. As a motivation, we start by showing how related work is prone to crash-attacks. The adversary learns time relations of log entries in the log, even if log entries are encrypted. That is, the time of an event determines the position of its log entry in the log file. This allows the adversary to, e.g., remove the most recent ℓ log entries, their integrity information, and then crash the log device.

In contrast, the idea of our new protocol SLiC is to (pseudo-)randomize positions of log entries in the log file. We design a new variation of Algorithm P by Knuth [7] to iteratively shuffle an array of an increasing number of log entries. Each of the n encrypted log entries will be at any position in the array with probability $\frac{1}{n}$. Thus, the adversary can only tamper or remove random log entries in the log, but not specific ones. The verifier, however, can reconstruct (pseudo-)random positions of each log entry, so they know which log entries can be lost in case of a device crash. Consequently, the verifier can distinguish regular crashes (the latest entries are lost) from crash-attacks (random entries are lost).

In summary, the technical highlights of this paper are:

- The first formal treatment of crash-attacks in the context of forward-secure logging. We demonstrate insecurity of recent work as soon as an adversary can deliberately crash the logging device.
- A new crash tolerant forward-secure logging protocol SLiC. To support either resource-constrained devices or large amounts of high-frequency log data, SLiC is very efficient and relies only on symmetric key cryptography. To add a new log entry, SLiC's time complexity is constant $O(1)$ in the total number of log entries. To recover a log of n log entries after a crash, its worst case complexity is $O(n \cdot \log n)$. SLiC is very general and does not require strong assumptions on underlying operating and file systems with respect to consistency.

- An *optimistic* version of our recovery technique for devices with small cache. Here, the complexity to recover n log entries is $O(n)$, i.e., asymptotically optimal.
- Besides a theoretical analysis where we formally prove its security, we also implement and evaluate SLiC practically. Our Python implementation realizes SLiC as a standard Syslogd server. On a 2.3 GHz i5, SLiC processes up to 700 syslog messages per second. This represents a slowdown by only one order of magnitude compared to just storing unprotected syslog messages.

2 Background

Based on initial ideas by Schneier and Kelsey [13], a significant amount of research has been conducted on forward-security and its application to secure logging, cf. [2, 4, 6, 10–12] and derivatives. These schemes focus on a symmetric key setting, and their main idea can be roughly summarized as follows.

Each *event* m_i, simply a bit string, is stored in the log file together with an authentication *tag* h_i as a log *entry*. For example for event m_i, a log entry $s_i = (m_i, h_i = \mathsf{HMAC}_K(m_i))$ would be added to the log file. To provide forward security, it is important that key K changes over time. Otherwise, if an adversary can compromise the device and learn key K, they would be able to modify old log entries. The rationale is therefore to change K for each log entry. If PRF is a pseudo-random function, key K_i can be computed as $\mathsf{PRF}_{K_{i-1}}(\chi)$ for some constant χ. To log event m_i, the logging device stores entry $(m_i, h_i = \mathsf{HMAC}_{K_i}(m_i))$, computes $K_{i+1} = \mathsf{PRF}_{K_i}(\chi)$, and then deletes K_i from its memory. As the verifier knows initial key K_0, they can reconstruct the chain of keys and verify individual log entries.

To protect against “truncation attacks” [10] where the adversary cuts the last ℓ entries from the log, related work adds a single, aggregated authentication tag H_i capturing the whole log file. For example, in addition to storing (m_i, h_i), the log device also stores $H_i = \mathsf{HMAC}_{K_i}(H_{i-1}, m_i)$ and deletes H_{i-1} for each log event m_i. If the adversary cuts entries from the end of the log, they would have to restore an old $H_j, j < i$.

Verification is straightforward: as the verifier can recompute all K_i, they can check the HMAC for each m_i, respectively. At the same time, they also iteratively compute the aggregated HMACs H_i. At the end of the log file, they check whether their aggregated H_n matches the one stored on the logging device.

Some work has targeted public verifiability using public-key cryptography [14]. Similar to other work using HMACs, they use signatures to protect log entries and signature aggregation to protect the whole log file.

None of the above work has investigated implications of (adversarial) log device crashes on log entry integrity.

Forward-Secure Logging and Crashes: Before formalizing security in the presence of crashes in the next section, we briefly demonstrate why the state of the art is insecure as soon as the adversary can crash the logging device (crash-attack).

Looking at current solutions above, we can identify that at least the following file system operations take place for each log event m_i.

```
1 store $(m_i, h_i)$;
2 store $H_i$;
3 delete $H_{i-1}$;
```

Many modern log-structured or journaling file systems such as YAFFS, EXT4, HFS or NTFS offer some guarantees regarding consistency of these three operations. However in this paper, we explicitly avoid making strong assumptions on the underlying storage (file system). Legacy systems, notably in industrial environments, use older file systems such as FAT32 or ext2. Some systems might even implement their own file systems, file system caches, and cache write-back strategies. The operating system, disk driver or even the disk itself can re-order disk writes for improved performance. A prominent example is the reordering of write operations following an elevator movement to minimize hard disk head seek times. If the three operations are reordered, and a crash occurs, this can lead to inconsistencies.

While there is exists a rich theory on consistency and hazards, we refrain from rigorous formalization at this point. The following more intuitive definition is sufficient for our security discussion: after logging event m_i, we want that *either* all operations 1 to 3 have been successfully performed, i.e., log entry (m_i, h_i) and H_i are stored on the disk, and H_{i-1} has been deleted, *or* none of the three operations has been performed.

This would allow the verifier to successfully verify the log, potentially with a missing last log entry. However by chance, it is possible that H_{i-1} is deleted from the disk, but H_i has not been (successfully) written. That is, file system operations 2 and 3 have been reordered by the operating system, and the logging device crashed after performing operation 3. Consequently, there is no valid H_i anymore on the disk, and log verification fails. Note that we are primarily concerned about disk write operations (store, delete). Disk reads are uncritical in our particular context.

A straightforward way to handle verification in case a logging device has crashed would be to accept a log of a crashed device even with H_i missing. Yet, this would allow adversary $\mathcal{A}$ compromising the log device to perform a crash-attack: $\mathcal{A}$ would truncate the log file, delete H_n, and then deliberately crash the device. Verifier $\mathcal{V}$ recovering the crashed log would not be able to distinguish this "truncation&crash" crash-attack from a regular crash. In conclusion, related work does not cope with crash-attacks.

We stress that this paper focuses on scenarios with the logging device being offline most of the time and no frequent connectivity to the verifier. This is the case in environments where network connectivity to the verifier is expensive or impossible, such as in industrial environments with unattended computer systems. Still, if an adversary can compromise the logging device, e.g., by physical access, we require some security guarantees. For completeness sake, we mention that if frequent connectivity with the verifier is available, other solutions become possible where the device can periodically offload some of its state for security, see Bowers et al. [5].

3 System and Adversary Model

We now present an overview over our system and adversary model. We envision a general scenario with three parties: 1) a logging device $\mathcal{L}$, 2) a verifier $\mathcal{V}$, and 3) an adversary $\mathcal{A}$. Logging device $\mathcal{L}$ receives log events $m_i \in \{0,1\}^*$ and writes them somehow to its storage.

At some point in time, $\mathcal{A}$ compromises $\mathcal{L}$. Informally speaking, by compromise we mean that $\mathcal{A}$ gets full access to log and internal state of the logging device. $\mathcal{A}$ can read out secrets, change the program, and tamper with the storage. $\mathcal{A}$ can even crash the logging device.

Finally, verifier $\mathcal{V}$ downloads the (crashed) log file from the logging device and checks the log's integrity. We consider scenarios where $\mathcal{L}$ logs autonomously and unattended. That is, besides an initial exchange of system parameters and keys, there is no communication channel between $\mathcal{L}$ and $\mathcal{V}$. Device $\mathcal{L}$ logs unattended by $\mathcal{V}$, and $\mathcal{V}$ gets access to $\mathcal{L}$'s log only after some time. If there would be a permanent communication channel, $\mathcal{L}$ could automatically forward all log events to $\mathcal{V}$.

Our security goal will be, roughly, that $\mathcal{A}$ cannot modify or remove log events from before the time of compromise without being detected.

3.1 Logging Protocols

A logging protocol Π is not required to be file-based and to append a single log entry for each event as related work. Very general, protocol Π comprises three algorithms:

1. $\mathsf{Gen}(1^{\lambda})$: Algorithm Gen takes security parameter λ as input and outputs Σ_0, $\mathcal{L}$'s initial *state*.
   $\mathcal{L}$'s *state* comprises all information currently stored on $\mathcal{L}$, e.g., a log file (initially empty), cryptographic keys etc. The initial state Σ_0 is shared by $\mathcal{L}$ and verifier $\mathcal{V}$.
2. $\mathsf{Log}(\Sigma_{i-1}, m_i)$: For new log event $m_i \in \{0,1\}^*$ and old state Σ_{i-1}, Log either outputs an updated state Σ_i, or a special state Σ_i^{Cr}. We call a state Σ_i^{Cr} a *crashed* state, and non-crashed states are simply *valid* states. With Σ_i^{Cr} we model a crashing $\mathcal{L}$. After algorithm Log has output a crashed state, it cannot be executed again; device $\mathcal{L}$ has crashed.
3. $\mathsf{Recover}(\Sigma, \Sigma_0)$: Receives as input either a valid state $\Sigma = (\mathsf{Log}(\mathsf{Log}(\ldots \mathsf{Log}(\Sigma_0, m_1)\ldots), m_n)$ or a crashed state Σ^{Cr}. A crashed Σ^{Cr} does not contain all n log events, but only $n' < n$. That is, $(n - n')$ events were *lost* due to a crash and are impossible to recover (e.g., hard disk power off during writing of some blocks).
   $\mathsf{Recover}$ outputs a set of $n' \leq n$ index-event tuples $\{(\tau_1, m'_1), \ldots, (\tau_{n'}, m'_{n'})\}$. For tuple (τ_i, m'_i), τ_i denotes the original index (order) of event m'_i. For example, if $\tau_3 = 5$, then m'_3 was the 5th log event m_5. We require $\mathsf{Recover}$ to unambiguously recover an event for a specific index; all indices τ must be different $(\forall i \neq j : \tau_i \neq \tau_j)$.
   In case of a crash, $\mathsf{Recover}$'s contribution is to use Σ and simply recover some of the original log events m_i. For correctness, we require that if Σ is a valid state, $\mathsf{Recover}$ outputs $\{(1, m_1), \ldots, (n, m_n)\}$. If Σ is a crashed state Σ^{Cr}, $\mathsf{Recover}$ outputs only a *subset* of $\{(1, m_1), \ldots, (n, m_n)\}$.
   Finally, if $\mathsf{Recover}$ detects that $\mathcal{A}$ has tampered with state Σ, it can also output special symbol $\perp$.

Algorithm Log is executed by device $\mathcal{L}$ and $\mathsf{Recover}$ by verifier $\mathcal{V}$. Note that $|\Sigma_n| \in \Omega(n)$, i.e., the size of the state has to be at least linear in the number of log events. Otherwise, administrator $\mathcal{V}$ would never be able to recover all log events from a state. As a result, $\mathsf{Recover}$'s time complexity must be in $\Omega(n)$, too. In conclusion, the idea behind $\mathsf{Recover}$ is that it can recover log events out of a crashed state Σ^{Cr}. Based on Σ^{Cr}, it should output a subset of all log events.

Efficiency: To support high frequency logging or resource-constrained hardware, we target Log's computational complexity to be constant in the number of log events n.

3.2 System Cache

A crash might not only involve a single log event not being correctly added to the log. Depending on the frequency of log events and the size and strategy of operating system and disk caches, many disk operations ($\mathsf{store}, \mathsf{delete}, \ldots$) can reside in the device's Cache. In case of a crash, only a random subset of these operations is executed.

Thus, we introduce an important system parameter: cache size cs. Depending on Cache (OS cache, filesytem cache, hard disk cache, ...), and the expected frequency of log events, you can roughly estimate an upper bound for the number cs of lost log events. This is the maximum number of log events that might not have been properly included in the log after a crash, because their disk write operations were residing in a cache. To support a broad range of real-world scenarios, we assume that, in case of a crash, the resulting disk operations of these cs events are executed 1) only partially, and 2) in a random order. Typically, cs is a constant system parameter, independent of the total number of log events n. Being a part of the device's state, we write $\mathsf{Cache} \subset \Sigma$.

Depending on the concrete logging protocol Π, each invocation of $\mathsf{Log}(m_i)$ creates multiple write operations. These write operations write data on disk that is necessary to later recover m_i. However in addition, $\mathsf{Log}(m_i)$ might also imply disk writes affecting verification of other events m_j. This leads to the following two definitions.

Definition 1 (Disk Write Operations). *Let m_i be a log event. A disk write operation $o(m_i)$ is a disk write of m_i's data necessary to later verify m_i.*

Let $\Pi = (\mathsf{Gen}, \mathsf{Log}, \mathsf{Recover})$ be a logging protocol. For event m_i, we define $\mathcal{O}^{\mathsf{Log}}(\Sigma_{i-1}, m_i) = \{o(m_u), \ldots, o(m_v)\}$ to be the set of disk write operations implied by adding m_i to the log with algorithm Log.

Basically, $\mathcal{O}^{\mathsf{Log}}$ is the set of log events m_j that is impacted by adding m_i to the log. If $\mathcal{L}$ crashes during $\mathsf{Log}(\Sigma_{i-1}, m_i)$ and $m_j \in \mathcal{O}^{\mathsf{Log}}(\Sigma_{i-1}, m_i)$, then m_j is also affected by the crash, as parts of its data might have been unsuccessfully written.

Therewith, we can now introduce the notion of expendable log events. A log event is expendable, if it might have been lost due to a crash.

Definition 2 (Expendable Log Event). *Let Σ_n be a valid state comprising events $\{m_1, \ldots, m_n\}$ and $\mathsf{Cache}_n = \varnothing$, and let $\mathsf{Cache}_{n'}$ be the contents of the cache after $\mathcal{L}$ adds events $(m_{n+1}, \ldots, m_{n'})$ with* Log. *An event m_i is* expendable *in state $\Sigma_{n'}$,* iff

$$o(m_i) \in \{\mathcal{O}^{\mathsf{Log}}(\Sigma_n, m_{n+1}) \cup \cdots \cup \mathcal{O}^{\mathsf{Log}}(\Sigma_{n'-1}, m_{n'})\} \wedge o(m_i) \in \mathsf{Cache}_{n'} \subset \Sigma_{n'}.$$

The set of all expendable log entries in $\Sigma_{n'}$ is $\mathcal{E} = \{m_i | m_i$ is expendable in $\Sigma_{n'}\}$.

The rationale of how $\mathcal{V}$ detects a crash-attack will be based on whether there are log entries missing that are not expendable at the time of the crash. As we will see, $\mathcal{V}$ can reconstruct indices of log events in Cache at the time of the crash.

Crash Functionality: To allow adversary $\mathcal{A}$ to learn about the implications of crashes on device states, we introduce another functionality $\mathsf{Crash}(\Sigma, \Pi)$ which $\mathcal{A}$ can call.

```
$\mathsf{Exp}^{\mathsf{CrInt}}_{\mathcal{A},\Pi,\mathsf{Crash}}(\lambda)$ :
1  $(m_1,\ldots,m_n,\mathrm{st}_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda, \mathsf{Gen}, \mathsf{Log}, \mathsf{Recover}, \mathsf{Crash})$;
2  $\Sigma_0 \leftarrow \mathsf{Gen}(1^\lambda)$;
3  for $i = 1$ to $n$ do
4    $\Sigma_i \leftarrow \mathsf{Log}(\Sigma_{i-1}, m_i)$
5  end
6  $(\Sigma', \alpha_1, \ldots, \alpha_\ell) \leftarrow \mathcal{A}(\mathrm{st}_{\mathcal{A}}, \Sigma_n, \mathsf{Gen}, \mathsf{Log}, \mathsf{Crash})$;
7  $\mathcal{R} \leftarrow \mathsf{Recover}(\Sigma', \Sigma_0)$ // Either $\mathcal{R} = \perp$ or $\mathcal{R} = \{(\tau_1, m'_1), \ldots, (\tau_{n'}, m'_{n'})\}$
8  if $\mathcal{R} = \perp$ then output $\perp$;
9  else if $\exists i \leq n' : m'_i \neq m_{\tau_i}$ then output forge;
10 else if $[\forall \alpha_i : m_{\alpha_i} \notin \mathcal{E}_n \wedge m_{\alpha_i} \notin \{m'_1, \ldots, m'_{n'}\}]$ then output delete;
11 else output $\{m'_1, \ldots, m'_{n'}\}$ endif
```

Experiment 1: Crash Integrity

The output of $\mathsf{Crash}(\Sigma, \Pi)$ is a crashed state Σ^{Cr} which would be the state of logging device $\mathcal{L}$ running Π and crashing at a time where its internal state is Σ. In practice, Crash reflects an adversary running a logging device on their own and playing with the effects of crashes.

3.3 Security Definition

The challenge for a security definition supporting crashes is that a crash implies losing a set of log events based on hardware and operating system properties. No security protocol can protect against such crashes. Instead, it should be difficult for $\mathcal{A}$ to delete specific events they choose independent of the crash.

We present our new security model in Experiment 1. In a first phase in Line 1, adversary $\mathcal{A}$ gets oracle access to $\Pi = (\mathsf{Gen}, \mathsf{Log}, \mathsf{Recover})$ and Crash functionalities. Oracle access allows $\mathcal{A}$ to learn about the system and prepare their attack.

After learning, $\mathcal{A}$ must output a sequence of log events m_i which challenger $\mathcal{L}$ will log. For each log event m_i, Log simply updates $\mathcal{L}$'s internal state. Eventually at time n, $\mathcal{A}$ compromises $\mathcal{L}$ and receives state Σ_n. Again, $\mathcal{A}$ gets oracle access, but only to functionalities Gen, Log, and Crash (see discussion below). Now, $\mathcal{A}$ outputs a tampered state Σ' together with a sequence of positions α_i. These α_i are positions of log events $\{m_{\alpha_1}, \ldots, m_{\alpha_\ell}\} \subseteq \{m_1, \ldots, m_n\}$ that $\mathcal{A}$ wants to remove from the log. To avoid trivial attacks, the m_{α_i} must not be expendable log events such as the ones currently in the cache. Expendable log entries would be lost anyways in a regular crash, so $\mathcal{A}$ could "legitimately" remove them just by crashing $\mathcal{L}$.

Based on Σ', algorithm Recover outputs a sequence of $n' \leq n$ index-event tuples $\{(\tau_i, m'_i)\}$. If among events $\{m'_1, \ldots, m'_{n'}\}$ there is at least one m'_i that differs from m_{τ_i}, then $\mathcal{A}$ has successfully forged (or modified) the τ_i^{th} event. If Recover outputs a sequence of events $(m'_1, \ldots, m'_{n'})$ that does not comprise $\{m_{\alpha_1}, \ldots, m_{\alpha_\ell}\} \subseteq \{m_1, \ldots, m_n\}$, then $\mathcal{A}$ has successfully deleted log events. We stress that $\mathcal{A}$ specifies which events to delete. Their goal is not to remove any element, but instead those that reveal $\mathcal{A}$'s attack.

Definition 3 (Crash Integrity). *A logging protocol* $\Pi = (\mathsf{Gen}, \mathsf{Log}, \mathsf{Recover})$ *provides* $[f_1(\cdot), f_2(\cdot)]$-crash integrity*, iff for all PPT adversaries* $\mathcal{A}$ *there exist functions* $f_1(\cdot)$ *and*

$f_2(\cdot)$ *such that*

$$Pr[\mathsf{Exp}^{\mathsf{CrInt}}_{\mathcal{A},\Pi,\mathsf{Crash}}(\lambda) = \mathsf{forge}] \leq f_1(\cdot) \ \wedge \ Pr[\mathsf{Exp}^{\mathsf{CrInt}}_{\mathcal{A},\Pi,\mathsf{Crash}}(\lambda) = \mathsf{delete}] \leq f_2(\cdot).$$

This very general security definition allows to upper-bound adversarial success probabilities. Ideally, functions f_1 and f_2 become very (negligible) small depending on their concrete input parameter, e.g., the number of events ℓ to delete or a security parameter λ. We give concrete examples in Section 4.

Discussion: As you can see from this definition, we now distinguish between two classes of attacks, forgery and deletion. We do this to allow for greater flexibility and support schemes where, e.g., function f_1 is significantly smaller than f_2, or f_1 uses different security parameters (as we will see later with SLiC). Obviously, there can be schemes with $f_1 = f_2$. Also note that $\mathcal{A}$ cannot adaptively choose log events m_i in $\mathsf{Exp}^{\mathsf{CrInt}}_{\mathcal{A},\Pi,\mathsf{Crash}}$. No scheme can be adaptively secure, as observing an intermediate Σ_i would allow $\mathcal{A}$ to later "rewind" Σ_n to Σ_i, by just presenting Σ_i in Line 6 of $\mathsf{Exp}^{\mathsf{CrInt}}_{\mathcal{A},\Pi,\mathsf{Crash}}$. As Σ_i has been a valid state, Recover would simply output a sequence of events, and tampering would go unnoticed.

Finally, note that the set of expendable log entries $\mathcal{E}$ depends on a concrete protocol Π. One might argue that there could be trivial protocols which, e.g., would rewrite the whole log file or database for each invocation of Log. Roughly speaking, all log events would be expendable leading to worthless security. However, we are interested in efficient protocols, specifically where Log has $O(1)$ computational complexity and cs is constant, too. This leads to an asymptotically constant number of expendable events which is small in practice and therewith meaningful security.

Our adversary model is similar to covert adversaries by Aumann and Lindell [1]. Adversary $\mathcal{A}$ is fully malicious, but wants to achieve a goal without being detected. You can imagine various real-world scenarios where $\mathcal{A}$ wants to keep a compromise undetected. For example, $\mathcal{A}$ has extracted sensitive information or wants to continuously spy on a system keeping it as future asset (even after the crashed device is rebooted).

4 Crash Recovery with SLiC

Based on our presentation of related work in Section 2, we make two observations. First, using an aggregated tag (HMAC or signature) over the whole log file is useless in the presence of crashes. Due to potential write reordering, the aggregated tag might be lost or not up to date after a crash. Second, with related work, even if log events m_i are encrypted, the adversary can easily correlate which log event corresponds to which log entry in the log file. Related work appends a new log entry to the log file, so the most recent log events correspond to the log entries at the end of the file. This helps the adversary to remove specific log entries, i.e., events, of their liking.

We will now present SLiC, a new protocol for crash integrity, and a modification of it that we call SLiC$^{\mathsf{Opt}}$. Both SLiC and SLiC$^{\mathsf{Opt}}$share the same idea of initialization and log algorithms, and they differ only in their recovery algorithm. To recover all n' log events from a (crashed) log, SLiC's Recover has $O(n' \cdot \log n')$ run time complexity, but does not rely on any additional assumption. In contrast, SLiC$^{\mathsf{Opt}}$is *optimistic* and assumes a log device $\mathcal{L}$ with cache or filesystem properties such that. SLiC$^{\mathsf{Opt}}$has time complexity $O(n')$; this is asymptotically optimal.

4.1 Overview

Before presenting technical details, we first give an overview of our main idea and ingredients to reach crash integrity.

To generally achieve forward integrity for previous log events, we can store for each log event m_i a log entry s_i consisting of m_i and h_i. As with related work, h_i is, e.g., $\mathsf{HMAC}_{K_i}(m_i)$. Again, key K_i is evolved from K_{i-1}, and K_{i-1} is thrown away. So, $\mathcal{A}$ cannot tamper and modify old log entries without being detected. This holds even if $\mathcal{A}$ can crash logging device $\mathcal{L}$. Therewith, we already achieve forward integrity.

However, the challenge for crash recovery is that we cannot rely on a protection mechanism securing integrity of the set of all log entries, i.e., completeness of the log as a whole. As shown before, using a simple *tag* would be prone to crashes which in return can be exploited by the adversary performing a truncation attack and rewinding the log to a previous version. Thus, we abstain from whole-log tag protection. Instead, our ideas is to randomize the mapping between log events and the position of their corresponding log entries in the log file. If $\mathcal{A}$ cannot determine which log entry corresponds to which event, it becomes difficult for them to change the log to a proper previous state where only expendable log entries are missing. Random modifications to the log by $\mathcal{A}$ will be detected by verifier $\mathcal{V}$ and allow to distinguish from real crashes.

Randomized Mappings: The mapping between an event m_i and a log entry s_j will be based on a PRG. Informally, this mapping will look like a "random" mapping to $\mathcal{A}$, but is deterministic to $\mathcal{V}$ as $\mathcal{V}$ knows the initial *seed* for the PRG. Similarly to evolving keys in related work after each log event, the PRG will be used in a forward-secure manner by updating its seed after each invocation.

To randomize mappings, we devise a new array shuffling technique based on Knuth's "Algorithm P" [7]. Instead of Knuth's random shuffle of a fixed-length array of n elements in place, our technique gradually builds a random shuffle of an array of increasing length. The idea is to swap a newly added element with an element at a random position in the array.

We apply this idea to the generation of a log file. We first compute the i^{th} log entry s_i by authentically encrypting m_i and then add s_i to the log by swapping it with a previous log entry, randomly chosen from a position between 1 and i. We show that for adversary $\mathcal{A}$ and a log file with n entries, the position of any entry is uniformly distributed. That is, at all times, a log entry in an array of current length n is at any position with probability $\frac{1}{n}$.

Recovery: To be able to recover log events from their random positions in the log, we augment the i^{th} log entry by $\kappa_i = \mathsf{PRF}_{K_i}(i)$. During recovery, we then sort all n log entries based on their κ_i values. To recover the i^{th} event m_i, we search in the sorted list of log entries for $\kappa_i = \mathsf{PRF}_{K_i}(i)$. This is the main idea of SLiC's $\mathsf{Recover}$.

An alternative way to recover log entries relies on the fact that verifier $\mathcal{V}$ knows the initial state of the PRG. Therefore, $\mathcal{V}$ can re-compute the random coins used during

```
Input: Security parameter $\lambda$
Output: Initial state $\Sigma_0$
$K_0 \xleftarrow{\$} \{0,1\}^\lambda$;
$seed_0 \xleftarrow{\$} \{0,1\}^\lambda$;
// Let $\mathcal{S}$ be a dynamic array, fill with random permutation
    of $\{dummy_1, \ldots, dummy_\lambda\}$ using Log
$\mathcal{S} \leftarrow \lambda$ randomly ordered dummy events;
output $\Sigma_0 = (K_0, seed_0, \mathcal{S})$;
```

Algorithm 2: Gen(λ)

swapping of entries and is able to determine which log entry resides in which position in the log. This is the idea of SLiC$^{\mathsf{Opt}}$'s Recover.

Security Rationale: The security rationale of this position randomization is that $\mathcal{A}$'s probability to successfully remove ℓ log events from a log is hypergeometrically distributed. While for very small ℓ, it is easy to remove log events, a larger ℓ implies increasing difficulty for $\mathcal{A}$ to remove all m_{α_i}. Still, a hypergeometric probability does not give much security, as $\mathcal{A}$ could also delete all but one log entry. We therefore require a certain minimum number of events in a log by initially adding *dummy* events. Moreover, $\mathcal{V}$ checks whether the set of recovered log events is *plausible*, i.e., all missing log events could have been lost in a crash.

The idea of SLiC's plausibility check is to verify whether all missing log events are expendable. Missing log events that are not expendable can only be due to adversarial tampering with the log: a crash-attack.

4.2 SLiC Details

First, to protect individual log events against modifications and forgery, we use standard authenticated encryption, e.g., encrypt-then-MAC as related work. That is, for a log event m_i, we prepare a log entry $s_i = (c_i, h_i)$ with $c_i = \mathsf{Enc}_{K_i}(m_i)$ and $h_i = \mathsf{HMAC}_{K_i}(c_i)$. We change key K_i to K_{i+1} after each log entry by computing $K_{i+1} = \mathsf{PRF}_{K_i}(\chi)$ for some constant χ. Similarly, after using a PRG with a seed $seed_i$, we update the seed to $seed_{i+1} = \mathsf{PRF}_{seed_i}(\chi')$. This already guarantees forward integrity.

Gen: Algorithm 2 shows SLiC's initialization Gen. Key K_0 and seed $seed_0$ are chosen uniformly from random. Also, a "dynamic" array $\mathcal{S}$ is created which will be used to store log entries. Dynamic simply says that the length of $\mathcal{S}$, i.e., the number of log entries stored in $\mathcal{S}$, can increase over time. Following standard notation, we write $\mathcal{S}[i]$ to point to the element at position i in $\mathcal{S}$. We initialize $\mathcal{S}$ by storing λ dummy events in a random order in it. To add these dummy elements $dummy_1, \ldots, dummy_\lambda$ to $\mathcal{S}$, we can use, e.g., the same idea than in our Log mechanism that we describe next. The output of Gen is the initial state Σ_0 comprising key K_0 seed $seed_0$, and array $\mathcal{S}$. State Σ_0 is shared between $\mathcal{L}$ and $\mathcal{V}$. Actually, it is sufficient to only share K_0 and $seed_0$ with $\mathcal{V}$ to reconstruct Σ_0.

Log: Algorithm 3 describes details of Log. First, we authentically encrypt new log event m_i and compute sorting key κ_i. As the current length of array $\mathcal{S}$ is $\lambda + i - 1$, we then

```
Input: Old state $\Sigma_{i-1}$, log event $m_i$
Output: Updated state $\Sigma_i$
// Let $\Sigma_{i-1} = (K_{i-1}, seed_{i-1}, \mathcal{S})$, $|\mathcal{S}| = \lambda + i - 1$
$c_i = \mathsf{Enc}_{K_{i-1}}(m_i)$; $h_i = \mathsf{HMAC}_{K_{i-1}}(c_i)$; $\kappa_i = \mathsf{PRF}_{K_{i-1}}(i)$;
$s_i = (c_i, h_i, \kappa_i)$;
$pos \xleftarrow{\$\mathsf{PRG}(seed_{i-1})} \{1, \ldots, \lambda + i\}$;
if $pos = \lambda + i$ then
    $\mathcal{S} = \mathcal{S}||(s_i)$;
else
    $\mathcal{S} = \mathcal{S}||\mathcal{S}[pos]$; $\mathcal{S}[pos] = s_i$;
end
$K_i = \mathsf{PRF}_{K_{i-1}}(\chi)$; $seed_i = \mathsf{PRF}_{seed_{i-1}}(\chi')$;
output $\Sigma_i = (K_i, seed_i, \mathcal{S})$;
```

Algorithm 3: $\mathsf{Log}(\Sigma_{i-1}, m_i)$

randomly select a position pos between 1 and $\lambda+i$. With $\mathsf{PRG}(seed_i)$ we denote that the random coins required to determine pos are based on a PRG with seed $seed_i$. Position pos is the position where we store new log entry $s_i = (c_i, h_i, \kappa_i)$. If $pos \neq \lambda + i$, we perform a swap: we do not append s_i to $\mathcal{S}$, but append the old contents of $\mathcal{S}[pos]$ to $\mathcal{S}$ and write s_i at $\mathcal{S}[pos]$. Finally, we evolve K_{i-1} to K_i and $seed_{i-1}$ to $seed_i$ and output the updated state Σ_i.

Recover: SLiC's Recover technique is shown in Algorithm 4. As input, this algorithm receives a possibly crashed state Σ containing some permutation π' of log entries $\pi'(s'_1, \ldots, s'_{n'})$. Being a potentially crashed state, some of the s'_i might be "broken", i.e., not fully written to disk and contain junk. Just the size of Σ allows storing of up to n' log entries. As we know the size of a log entry, we can therefore parse n' potentially broken log entries out of Σ.

First, we advance K_0 and $seed_0$ to the earliest possible time of a valid state. If Σ has n' entries and cache size is cs, then at least $n' - \mathsf{cs}$ log events were properly added to the log, and up to cs events might have been in the cache or only partially written to the log. We then prepare the set of expendable log events $\mathcal{E}$ (see more details later in Section 4.3). Yet, to be able to compute $\mathcal{E}$, we reconstruct the permutation π that determines where log entries should be located in a valid $\mathcal{S}$. Here, $\pi[i] = j$ denotes that log entry s_i resides at $\mathcal{S}[j]$. Permutation π^{-1} is the inverse permutation to π.

It is straightforward to compute π and π^{-1} as $\mathcal{V}$ can replay $\mathcal{L}$'s random coins and therewith the swaps. $\mathcal{V}$ knows the initial seed $seed_0$, and for each pos computed in Line 8 of Algorithm 4, π and π^{-1} can be updated in constant time. The updated π and π^{-1} then allow adding the next two expendable log entries to $\mathcal{E}$. As we do not know when exactly the crash occurred, we iterate over all possible times of the crash, i.e., between the $(n' - \mathsf{cs})^{\text{th}}$ and $(n' + \mathsf{cs})^{\text{th}}$ event.

We sort log entries s'_i by their keys κ'_i and store them in a binary search tree. This allows to search for individual entries in logarithmic time. We iterate over all $n' + \mathsf{cs}$ possible log entries that could have resided in a crashed state of length n' (we will see later why) and check their HMACs. If the HMACs match, we add the individual log entry to set $\mathcal{R}$. Finally, we perform a plausibility check to distinguish a regular crash from a crash-attack. The total number of log entries recovered must be at least $\lambda - \mathsf{cs}$,

```
Input: State $\Sigma$ to check, initial state $\Sigma_0$
Output: Recovered log events $\{m_1, \ldots, m_{n'}\}$
// Let $\Sigma_0 = (K_0, seed_0, \mathcal{S}_0)$
// Parse $\Sigma$ as $(K_{n'}, seed_{n'}, \mathcal{S}' = \pi'(s'_1, \ldots, s'_{n'}))$, let $s'_i = (c'_i, h'_i, \kappa'_i)$
$\mathcal{R} = \varnothing; \mathcal{E} = \varnothing;$
// Evolve key, seed, $\pi, \pi^{-1}$
for $i = 1$ to $n' - \mathsf{cs} - 1$ do
    $K_i = \mathsf{PRF}_{K_{i-1}}(\chi);\ seed_i = \mathsf{PRF}_{seed_{i-1}}(\chi');$
    Update $\pi$ and $\pi^{-1}$;
end
// Compute expendable log indices
for $i = n' - \mathsf{cs}$ to $n' + \mathsf{cs}$ do
    $K_i = \mathsf{PRF}_{K_{i-1}}(\chi);\ seed_i = \mathsf{PRF}_{seed_{i-1}}(\chi');$
    $pos \xleftarrow{\$\mathsf{PRG}(seed_i)} \{1, \ldots, i\};$
    Update $\pi$ and $\pi^{-1}$;
    $\mathcal{E} = \mathcal{E} \cup \{i, \pi^{-1}[pos]\};$
end
// Sort log entries based on $\kappa'_i$
$\mathsf{SearchTree} = \mathsf{Sort}(\pi'(s'_1, \ldots, s'_{n'}));$
for $i = 1$ to $n' + \mathsf{cs}$ do
    $\kappa_i = \mathsf{PRF}_{K_i}(i);$
    $(c'_i, h'_i) = \mathsf{BinSearch}(\mathsf{SearchTree}, \kappa'_i);$
    if $\mathsf{HMAC}_{K_i}(c'_i) = h'_i$ then
        $\mathcal{R} = \mathcal{R} \cup \{(i, \mathsf{Dec}_{K_i}(c'_i))\}$
    end
end
// Check plausibility
if $(|\mathcal{R}| < \lambda - \mathsf{cs}) \vee (\exists i \in \{1, \ldots, n' - \mathsf{cs}\} : \{(i, \cdot)\} \notin \mathcal{R} \wedge i \notin \mathcal{E})$ then output $\perp$ ;
else output $\mathcal{R}$ endif;
```

Algorithm 4: $\mathsf{Recover}(\Sigma, \Sigma_0)$

i.e., the number of dummy elements that were in the log initially minus the size of the cache. For a potentially crashed state of length n', we know that at least entries $m_i - \mathsf{cs}$ were once written into $\mathcal{S}$. So, if we cannot recover a log event $m_i, 1 \leq n' - \mathsf{cs}$, it must be in the set of expendable events $\mathcal{E}$. If not, we know that $\mathcal{A}$ has removed it.

Overwriting keys: As mentioned in Section 3.1, a state Σ contains all information currently stored in $\mathcal{L}$, in our case including cryptographic keys and seeds. So far, we have ignored that when we overwrite K_{i-1} by K_i in Algorithm 3, the disk operation is buffered in the cache, too. One might argue that over time many old K_i (and seeds) remain in the cache before being evicted, allowing $\mathcal{A}$ to successfully rewind. In practice, the situation is much simpler. Caches typically *replace* a write operation in a cache by a new one to the same location. Thus, $\mathcal{A}$ would be able to only recover the previous to the current key. Moreover, in our case neither keys nor seeds need to be persistent. That is, in contrast to aggregated authentication tags H of related work, they are not required to be written to disk, but can be lost in a crash. As a result, we can store them in main memory, where overwrites are instantaneous.

Algorithm 4 does not make use of $K_{n'-1}$ or $seed_{n'-1}$. Similar to (aggregated) authentication tags H of related work, these might be old or invalid due to a crash.

4.3 Complexity Analysis

Efficiency: Adding a new log event with Log has $O(1)$ computational complexity. Writing the new s_i to array $\mathcal{S}$ requires only a constant number of disk operations (two), even if $\mathcal{S}$ is realized as a simple file, and all m_i have the same length. If in practice the m_i have different lengths, we can simply pad them to a maximum size. Real-world log services such as syslogD specify a maximum size, e.g., 1024 Byte [9].

To estimate the computational complexity of $\mathsf{Recover}$, we inspect the 3 for-loops, Sort, and $\mathsf{BinSearch}$. The first for-loop has asymptotic run time of $O(n')$, and the second loop has constant $O(1)$ complexity (cs is a system constant). Sort runs in time complexity $O(n' \cdot \log n')$. As $\mathsf{BinSearch}$ has complexity $O(\log n')$ the third for-loop also has a complexity of $O(n' \cdot \log n')$. The plausibility check has a run time of $O(n')$ In total, $\mathsf{Recover}$'s time complexity adds up to $O(n \cdot \log n)$.

Computation of Expendable Log Events: Before analyzing security, we explain which elements belong into $\mathcal{E}$ and compute the number of expendable events $|\mathcal{E}|$.

Assume that $\mathcal{L}$ crashes while its cache is full. From a log file consistency perspective, this is the worst situation. So, there are cs events that have not been successfully written to the log file. Furthermore, assume that the size of $\mathcal{S}$ on disk at the time of the crash is n' events, some of them potentially broken. If we look at the swap operation in Algorithm 3, lines 4 to 8, there are two cases: either, s_i is just appended to $\mathcal{S}$ (single disk write operation, Definition 1), or an old s_j is first read from disk into the cache, and then s_i and s_j are written to disk. Consequently, the cache might contain a mix of previous log entries already written to $\mathcal{S}$ and new ones to be added.

In one extreme case, the cache only contains new entries, i.e., $(s_{n'+1}, \ldots, s_{n'+\mathsf{cs}})$. Therefore, these entries are expendable and belong to $\mathcal{E}$. The other extreme case is where over time all all entries before $s_i, i \leq n'$ have been read from disk, and cs new entries have been written to the old positions. That is, the cache contains cs old entries that are determined by the PRG. Consequently, also these old entries have to be in $\mathcal{E}$.

Any other cache configuration will contain just a mix of subsets of cs old and cs new log entries. In conclusion, $\mathcal{E}$ contains cs old entries and cs new entries, $|\mathcal{E}| = 2 \cdot \mathsf{cs}$.

4.4 Security Analysis

For space reasons, we have moved our security analysis to Appendix A. Here, we only state main results.

Theorem 1. *Let $\mathcal{A}$ know Σ_n comprising $\mathcal{S}$, $|\mathcal{S}| = n + \lambda$, let ℓ be the number of events $\mathcal{A}$ wants to delete,* cs *the system's cache size, and n' the number of entries output by $\mathcal{A}$ as part of their malicious state Σ'. Let $\epsilon(\lambda) = \max(\epsilon_{\mathsf{PRF}}(\lambda), \epsilon_{\mathsf{PRG}}(\lambda), \epsilon_{\mathsf{Enc}}(\lambda))$.*

For security parameter λ, if PRG *is a pseudo-random generator,* PRF *is a pseudo-random function, $|m_1| = \ldots = |m_n|$, and* Enc *is IND-CPA encryption, then SLiC* *provides* $[\epsilon_{\mathsf{PRF}}(\lambda), f(n, n', \ell, \mathsf{cs}, \lambda)]$*-crash integrity, with*

$$f = \begin{cases} 0 & \text{, } \textit{if } n' < \lambda - \mathsf{cs} \\ \max(\epsilon(\lambda), \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}}) & \text{, } \textit{otherwise.} \end{cases}$$

To understand implications of Theorem 1, in particular $\mathcal{A}$'s success probability for typical system parameters, we provide an asymptotic bound for the fraction of two binomial coefficients $\frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}}$ of f. Roughly speaking, the following Corollary 1 (proof in Appendix A) states that, as $\lambda < n$, f decreases exponentially in both ℓ and λ.

Corollary 1.

$$\exists n_0, \lambda_0 \textit{ s.t. } \forall n > n_0, \forall \lambda > \lambda_0 : \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}} < \max(e^{-2\cdot(\ell-\mathsf{cs})}, (\frac{\lambda}{n})^{\lambda} \cdot \frac{1}{n^{\ell-4\cdot\mathsf{cs}}}).$$

Practical Implications: Cache size cs can easily be upper bounded, if the maximum frequency of log events is known. During our experiments with syslogD on a Debian Linux laptop with 2.3 GHz i5, syslog UDP packets were dropped as soon as the rate was more than 500 events per second. In addition, a standard Linux kernel evicts a page cache entry after at most 30 sec [8]. So, with maximum rate r and eviction time t, $\mathsf{cs} \leq r \cdot t$. This back-of-an-envelope computation assumes the amount of RAM available for the page cache to be sufficient to hold $r \cdot t$ entries.

As of Corollary 1, f decreases exponentially with increasing security parameter λ and the number ℓ of elements $\mathcal{A}$ wants to delete. To support our theoretical security results with real-world parameters, we set the cache eviction time t to Linux' standard value of 30 sec, rate r to 500, $\lambda = 2^{15} = 32768$, $n = 33000$ (λ dummy entries plus a few "real" entries), and n' to the minimum of 2^{15}. Therewith, $\mathcal{A}$ can successfully delete a single ($\ell = 1$) unwanted entry with only probability $\approx 2^{-36}$. Removing $\ell = 10$ unwanted entries is possible with probability 2^{-100}. Even if $\mathcal{A}$ presents a state with $n' = n - 1$ entries and wants to remove $\ell = 1$ unwanted entries, their success probability in this configuration is quite low with $\approx 2^{-15}$. If n increases, e.g., $n = 2^{15} + 2^{10}$ (2^{10} real entries), $\ell = 10, n' = n - \ell$, success chances get very low with 2^{-129}. So in general, an increasing number of entries strengthens security. In practice, parameter λ renders adversarial success very small.

Remark: As all related work on forward-secure logging, we also assume that unwanted events m_{α_i} have been evicted from the system's cache at the time of the compromise. If an adversary has fully compromised a system, they could otherwise tamper with the operating system and remove cache entries containing the m_{α_i}. If $\mathcal{A}$ compromises $\mathcal{L}$, they can theoretically remove all events from the last 30 s (maximum cache entry eviction time). Without changing standard cache write-back behavior, no protocol for forward-secure logging can protect against these attacks. Although out of scope for this paper, one could imagine that, e.g., some small integrity information is allowed to circumvent the page cache and is directly written to disk.

4.5 Evaluation

To demonstrate its real world applicability, we have implemented and practically evaluated SLiC in Python. Our implementation uses AES-256 (CTR mode, PKCS#5 padding) and HMAC-SHA256 as underlying cryptographic primitives.

Besides SLiC's core functionality, we have implemented a wrapper. This wrapper allows SLiC's logging with either a text file as a source for log events, or register as a real syslog server. Other syslog servers, either on the same physical machine or remotely, can then send a copy of their syslog events to our secure logging syslog server. Our implementation accepts any text string received on its UDP port as a new log event, so it is trivially compatible with current versions of syslog, syslog-ng, rsyslog etc. While the UDP scenario with our protocol running as part of a real world syslog environment is more realistic, we also include the performance measurement on a file to avoid potential network and network stack side effects.

Our benchmark hardware has been 1) a Windows laptop with 2.6 GHz i5-2540M CPU, and 2) a Raspberry Pi B+ with 700 MHz ARM CPU running Linux. We set $\lambda = 2^{15}$ for good security. Computing $\lambda = 2^{15}$ dummy elements on the laptop takes $\approx$40 s ($\approx$770 dummy elements/s) and $\approx$1000 s on the Raspberry ($\approx$30 dummy elements/s). Computation time of dummy elements is linear in the number of dummy elements (but their number has an exponential effect on security, see security analysis in Appendix A). Note that dummy elements need to be computed only initially and once per log device.

File Benchmark: We prepare a file with 2^{20} random "log events", each a text string of length 160 characters. On our laptop, we can log $\approx$740 events/s. This represents a slowdown by a factor of 20 compared to simply storing plain, unprotected syslog events. The Raspberry Pi B+ can log $\approx$30 events/s, a slowdown by factor of 60.

As part of Recover, sorting all 2^{20} log entries is very fast and takes only $\approx$7 s on the laptop ($\approx$140000 entries/s). During subsequent recovery, $\approx$1000 log entries are processed/s. Finally, security checks are required to see, e.g., whether missing entries are expendable, cf. Algorithm 4. These plausibility checks are also fast, and the laptop performs them with $\approx$12000 log entries/s. On the Pi, sorting 2^{20} entries for recovery takes $\approx$1 min ($\approx$16000 entries/s). During recovery, the Pi processes $\approx$30 entries/s and performs plausibility checks with $\approx$320 entries/s.

UDP Benchmark: To measure UDP performance, we generated syslog events with a Python script on another machine and sent them to our SLiC syslog servers on laptop and Pi. In this setup, the laptop securely logs $\approx$500 syslog events/s and the Raspberry Pi $\approx$10 syslog events/s. Compared to the laptop, we conjecture the bad Pi performance to be caused by Python's poor I/O handling and writing to an SD card.

On both Linux and Windows, we experienced syslog UDP rate limitation (ca. 500/s), and packets were dropped. While Windows silently discards some UDP packets to cope with a receiving thread unable to process another UDP packet, the Linux kernel generates another syslog event to inform about a new rate limit. In an environment where actual delivery of a large amount of syslog events is important, one might consider changing the regular syslog communication protocol from UDP to TCP.

Although our prototypical implementation in Python is not optimized for performance and serves only as a proof of concept, we can conclude that SLiC is efficient.

Being able to log 500 (or even 700) log events per second on simple laptop hardware demonstrates usability in larger systems and extensive amounts of logging data. Even running on the resource-constrained Raspberry Pi as a remote logging server, SLiC can serve as a secure logging device in environments with fewer log events.

4.6 SLiC$^{\mathsf{Opt}}$

We briefly sketch an alternative to SLiC that we call SLiC$^{\mathsf{Opt}}$and that has optimal computational complexity $O(n')$. In many real-world scenarios, SLiC$^{\mathsf{Opt}}$is expected to run faster than SLiC in scenarios where cache size cs is smaller than $O(\log n)$. This is especially the case for RAM-constrained systems, systems that frequently evict their cache or a low rate of log entries.

Overview: SLiC$^{\mathsf{Opt}}$does not follow the approach of sorting log entries using a sorting key κ and subsequently searching for log entries. Instead, the idea is to directly find all non-expendable log entries at the positions in $\mathcal{S}$ where they are expected to reside. As $\mathcal{V}$ knows the initial keys and PRG seed, it can replay all random coins and therewith all permutations π. Given a potentially crashed state $\mathcal{S}$ with n' log entries, some of them may be broken, $\mathcal{V}$ can compute the permutation π at the time of the crash. Therewith, $\mathcal{V}$ can simply lookup all non-expendable entries and check whether their HMAC matches.

More precisely, as the crash might have occurred anywhere between time $(n' - \mathsf{cs})$ and $(n' - \mathsf{cs})$, $\mathcal{V}$ iterates over all $2 \cdot \mathsf{cs}$ possible permutations and tries to find all non-expendable log entries.

Details: We keep Gen and Log from SLiC and only present our modified $\mathsf{Recover}$ in Algorithm 5. Due to space reasons, he have moved Algorithm 5 to Appendix B. The main difference here is that $\mathcal{V}$ iterates over two nested loops. In the outer loop, $\mathcal{V}$ tries to recover the maximum $(n' + \mathsf{cs})$ of possible log entries. In the inner loop, $\mathcal{V}$ iterates over the possible times of a crash, i.e., between $(n' - \mathsf{cs})$ and $(n' + \mathsf{cs})$. In contrast to the previous algorithm for recovery, $\mathcal{V}$ must now keep track of expendable and recovered entries per possible crash time, i.e., $\mathcal{E}_j$ and $\mathcal{R}_j$. During each iteration, $\mathcal{V}$ tries to directly find the c' following the current permutation π. Permutations π are updated as before.

The plausibility check at the end verifies whether there was a time of crash j such that entries in $\mathcal{S}$ at time j match the plausibility check of Algorithm 4. That is, if there is no such time of crash j that would pass the plausibility check of Algorithm 4, then $\mathcal{V}$ has detected a crash-attack.

Run time of the outer loop is $O(n')$. As cs is a constant parameter, independent of the total number of log entries, the inner loop has run time $O(1)$. The plausibility check is over a constant number of possible crash times, and has run time of $O(n')$, too. In total, Algorithm 5 has optimal run time complexity $O(n')$.

5 Conclusion

In case a logging device crashes, log file and integrity information become inconsistent. An adversary compromising the device can exploit this by performing a crash-attack where they delete unwanted log entries and crash the device to hide attack traces. In this paper, our first contribution has been to show that previous work on forward-secure logging against compromising adversaries has not been designed with crash-attacks in mind and is insecure. Our second and main contribution is SLiC, a first forward-secure logging protocol that copes with crashes. Observing that any integrity tag for the log as a whole is futile, our idea is to decorrelate the time of a log event with its position in the log file. We encrypt and place log events at random positions in the log. Deleting specific log entries, therefore, becomes difficult for the adversary, and tampering is likely to be detected. We finally show that SLiC is efficient both asymptotically and practically. SLiC can process ≈700 log events per second even on simple laptop hardware.

References

[1] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010. ISSN 0933-2790.

[2] M. Backes, C. Cachin, and A. Oprea. Secure Key-Updating for Lazy Revocation. In *Proceedings of European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 327–346, Hamburg, Germany, 2006. Springer. ISBN 3-540-44601-X.

[3] M. Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. Cryptology ePrint Archive, Report 2006/043, 2006. `http://eprint.iacr.org/`.

[4] M. Bellare and B.S. Yee. Forward-Security in Private-Key Cryptography. In *Proceedings of Cryptographers' Track at the RSA Conference*, pages 1–18, San Francisco, USA, 2003. ISBN 978-3-540-36563-1.

[5] K.D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In *Proceedings of Research in Attacks, Intrusions and Defenses*, pages 46–67, 2014.

[6] J.E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of Australasian Symposium on Grid Computing and e-Research*, volume 54, pages 203–211, Hobart, Australia, 2006.

[7] D.E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*, volume 2, chapter 3.4.2, pages 139–140. Addison Wesley, 2nd edition, 1981. ISBN 978-0201896848.

[8] Linux Kernel Documentation. `/proc/sys/vm/dirty_expire_centisecs`, 2015. Standard value is 30 sec on kernel 3.16.0-4, 64 bit, `https://www.kernel.org/doc/Documentation/sysctl/vm.txt`.

[9] C. Lonvick. The BSD syslog Protocol. IETF, Request for Comments: 3164, 2001. `https://tools.ietf.org/html/rfc3164`.

[10] D. Ma and G. Tsudik. A New Approach to Secure Logging. *ACM Transactions on Storage*, 5(1), 2009. ISSN: 1553-3077.

[11] G.A. Marson and B. Poettering. Practical Secure Logging: Seekable Sequential Key Generators. Cryptology ePrint Archive, Report 2013/397, 2013. `http://eprint.iacr.org/`.

[12] G.A. Marson and B. Poettering. Even More Practical Secure Logging: Tree-Based Seekable Sequential Key Generators. In *Proceedings of European Symposium on Research in Computer Security*, volume 8713 of *Lecture Notes in Computer Science*, pages 37–54, Wroclaw, Poland, 2014. Springer. ISBN 978-3-319-11211-4.

[13] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.

[14] A.A Yavuz, P. Ning, and M.K. Reiter. BAF and FI-BAF: Efficient and Publicly Verifiable Cryptographic Schemes for Secure Logging in Resource-Constrained Systems. *Transactions on Information System Security*, 15(2):9, 2012. ISSN 1094-9224.

A Security Analysis

Lemma 1. *Let X_i be the random variable describing the position of log entry s_i in the log. If the output of* PRG *is pseudo-random, then after adding n events to the log using Algorithm 3:*

$$\forall i, j, 1 \leq i, j \leq n : Pr[X_i = j] = \frac{1}{n}.$$

Proof. We prove by induction over n.

1. Basis: Let $n = 2$. The log contains the two entries s_1 and s_2. When s_2 was added to the log, Algorithm 3 swapped s_1 with s_2 with probability $\frac{1}{2}$ (if PRG is pseudo-random). So, both log entries are at any of the two positions with probability $\frac{1}{n} = \frac{1}{2}$.
2. Inductive step: Let the claim be true for a log of length n entries. We now show that it also holds when adding the next log entry $n + 1$. To compute the probability that any log entry s_i is at any position j, we consider two cases.
   First, $\forall i, 1 \leq i \leq n + 1 : Pr[X_i = n + 1] = \frac{1}{n+1}$. That is, the probability that any of the $n + 1$ log entries is at position $n + 1$ is $\frac{1}{n+1}$, because Algorithm 3 selects any of them pseudo-randomly with equal probability.
   Second, we compute the complementary probability that any of the $n+1$ log entries is at a position $j, 1 \leq j \leq n$, left of $n + 1$.
   For entry s_{n+1}, $\forall j \leq n : Pr[X_{n+1} = j] = \frac{1}{n+1}$, as Algorithm 3 selects pseudo-randomly between 1 and $n + 1$. For the other entries, we have

   $$\begin{aligned}\forall i, &1 \leq i \leq n, \forall j \leq n : \\ &Pr[X_i = j] = Pr[s_i \text{ was at position } j \text{ before adding element } n+1 \\ &\wedge \text{ Algorithm 3 does not swap } s_i \text{ to position } n+1] \\ &= \frac{1}{n} \cdot \left(1 - \frac{1}{n+1}\right) = \frac{n+1-1}{n_0^2 + n} = \frac{1}{n+1}.\end{aligned}$$

   The first probability, s_i was at position j before adding element $n + 1$ (in Algorithm 3), equals $\frac{1}{n}$ by induction hypothesis.

□

For simplicity, we assume that $\mathsf{HMAC}_K(c) = \mathsf{PRF}_K(c, 1)$ in our analysis, where "," is an unambiguous pairing of inputs [3]. So, we use the same pseudo-random function to evolve keys and compute authentication tags.

Lemma 2. *Let $\mathcal{A}$ compromise $\mathcal{L}$ at time t when n events have been added to the log. Let $\mathcal{A}$ get access to internal state Σ_n comprising $\mathcal{S}$, $|\mathcal{S}| = n+\lambda$. If* PRG *is a pseudo-random generator,* PRF *is a pseudo-random function, and* Enc *is IND-CPA encryption, then the distribution of log entries $s_i, i \leq n$ in $\mathcal{S}$ logged before time t is indistinguishable from a random distribution for $\mathcal{A}$.*

Proof (Sketch). If PRF is a pseudo-random function, then $seed_t$ does not reveal details about seeds $seed_i$, keys K_i, and sort keys $\kappa_i, i < n$ to $\mathcal{A}$. As moreover PRG is a pseudo-random generator, this implies that $\mathcal{A}$ does not learn any information about previous swap operations during Log. Finally, with Enc being IND-CPA encryption, s_i does not leak information about m_i (even though $\mathcal{A}$ specified the distribution of m_is).

In conclusion, for $\mathcal{A}$ the distribution of s_i is pseudo-random. $\mathcal{A}$ cannot determine which event m_i (and entry s_i) is stored at which position in $\mathcal{S}$. □

In the following, we come back to Definition 3 and state our main security claim. Let $\epsilon_{\mathsf{PRF}}(\lambda), \epsilon_{\mathsf{PRG}}(\lambda), \epsilon_{\mathsf{Enc}}(\lambda)$ denote the negligible adversarial success probabilities of PRF, PRG, and Enc security when initialized with security parameter λ.

Theorem 1. *Let $\mathcal{A}$ know Σ_n comprising $\mathcal{S}, |\mathcal{S}| = n + \lambda$, let ℓ be the number of events $\mathcal{A}$ wants to delete,* cs *the system's cache size, and n' the number of entries output by $\mathcal{A}$ as part of their malicious state Σ'. Let $\epsilon(\lambda) = \max(\epsilon_{\mathsf{PRF}}(\lambda), \epsilon_{\mathsf{PRG}}(\lambda), \epsilon_{\mathsf{Enc}}(\lambda))$.*

For security parameter λ, if PRG *is a pseudo-random generator,* PRF *is a pseudo-random function, $|m_1| = \ldots = |m_n|$, and* Enc *is IND-CPA encryption, then SLiC provides $[\epsilon_{\mathsf{PRF}}(\lambda), f(n, n', \ell, \mathsf{cs}, \lambda)]$-crash integrity, with*

$$f = \begin{cases} 0 & , \textit{if } n' < \lambda - \mathsf{cs} \\ \max(\epsilon(\lambda), \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}}) & , \textit{otherwise}. \end{cases}$$

Proof (Sketch). We focus on delete attacks and quickly ignore forge attacks by stating that they are as likely as breaking HMAC security, i.e., $\epsilon_{\mathsf{PRF}}(\lambda)$.

First, if $n' < \lambda - \mathsf{cs}$, then there are less entries in $\mathcal{S}$ than theoretically possible. The initial size of $\mathcal{S}$ is λ, and at most cs entries of these can become expendable. So, $\mathcal{V}$ knows that $\mathcal{A}$ has removed an entry from $\mathcal{S}$.

In case, $n' \geq \lambda - \mathsf{cs}$, probability f is hypergeometrically distributed. Assume $\mathcal{A}$ compromises $\mathcal{L}$ at time $t = n - \lambda$, so there are a total of n events in $\mathcal{S}$. To succeed in the security experiment, $\mathcal{A}$ must present a subset of n' out of all n events such that none of the ℓ unwanted events is in this subset. For simplicity, we assume for now that the ℓ unwanted events are all expendable. Moreover, $\mathcal{A}$'s subset must contain all of the $(n' - 2 \cdot \mathsf{cs})$ non-expendable events. So, the probability computes to $\frac{\binom{\ell}{0}\cdot\binom{n'-2\cdot\mathsf{cs}}{n'-2\cdot\mathsf{cs}}\cdot\binom{n-\ell-n'+2\cdot\mathsf{cs}}{n'-n'+2\cdot\mathsf{cs}}}{\binom{n}{n'}} = \frac{\binom{n-\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}}}{\binom{n}{n'}}$.

A last requirement to succeed in the security experiment is that all ℓ unwanted events are part of the $2{\cdot}\mathsf{cs}$ expendable events. So, f computes to $\frac{\binom{\ell}{0}\cdot\binom{n'-(2\cdot\mathsf{cs}-\ell)}{n'-(2\cdot\mathsf{cs}-\ell)}\cdot\binom{n-\ell-n'+2\cdot\mathsf{cs}-\ell}{n'-n'+2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}} = \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}}$.

Finally, the adversary might win by breaking cryptographic tools used in SLiC. They can do that with probability $\epsilon(\lambda)$, such that f computes to $f = \max(\epsilon(\lambda), \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}})$. □

Again, we assume in this proof same-length log events m_i. As discussed in Section 4.3, we can pad different length events m_i to a maximum length, e.g., as standardized by an actual logging service [9].

Corollary 1.

$$\exists n_0, \lambda_0 \textit{ s.t. } \forall n > n_0, \forall \lambda > \lambda_0 : \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}} < \max(e^{-2\cdot(\ell-\mathsf{cs})}, (\frac{\lambda}{n})^{\lambda} \cdot \frac{1}{n^{\ell-4\cdot\mathsf{cs}}}).$$

Proof. Let $\mathsf{BIN} = \frac{\binom{n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell}}{\binom{n}{n'}}$. We bound BIN using standard binomial inequalities leading to $\mathsf{BIN} < g = \frac{(\frac{e\cdot(n-2\cdot\ell-n'+2\cdot\mathsf{cs}}{2\cdot\mathsf{cs}-\ell})^{2\cdot\mathsf{cs}-\ell}}{(\frac{n}{n'})^{n'}}$. Considering g as a function of n', we first compute its derivative and obtain

$$g'(n') = [(n-n'+2\cdot\mathsf{cs}-\ell)\cdot(1+\ln\frac{n'}{n}) - (2\cdot\mathsf{cs}-\ell)]\cdot(\frac{n}{n'})^{n'}\cdot A,$$

where A is a positive constant which neither depends on n nor n'.

If $n' > \frac{n}{e}$, then $(1 + \ln\frac{n'}{n})$ is strictly positive, and therefore there exists n_0 such that for $n > n_0$, $(n-n'+2\cdot\mathsf{cs}-\ell)(1+\ln\frac{n'}{n}) > (2\cdot\mathsf{cs}-\ell)$. Thus for $n > n_0$, we get $g'(n') > 0$ and conclude that g will be monotonically increasing in n'. Therefore, g, and consequently BIN, is *upper* bounded at the highest legitimate value n' that the adversary can select, i.e., $n' = n-\ell$. Alternatively, for $n' < \frac{n}{e}$, $\exists n_0$ such that $n > n_0$, $g'(n') < 0$. Therefore, g, is monotonically decreasing and consequently BIN, has an *upper* bound at the smallest legitimate value for n', i.e., $n' = \lambda - 2\cdot\mathsf{cs}$. So, $\mathsf{BIN} < \max(g(n'=n-\ell)$, $g(n'=\lambda-2\cdot\mathsf{cs}))$. We bound $g(n'=n-\ell)$ by

$$g(n'=n-\ell) = \frac{e^{2\cdot\mathsf{cs}-\ell}}{(\frac{n}{n-\ell})^{n-\ell}} < e^{-2\cdot(\ell-\mathsf{cs})}.$$

Similarly, we bound $g(n'=\lambda-2\cdot\mathsf{cs})$ by

$$\begin{aligned} g(n'=\lambda-2\cdot\mathsf{cs}) &< \frac{(e\cdot(n-2\cdot\ell-\lambda+4\cdot\mathsf{cs}))^{2\cdot\mathsf{cs}-\ell}\cdot(\lambda-2\cdot\mathsf{cs})^{(\lambda-2\cdot\mathsf{cs})}}{n^{\lambda-2\cdot\mathsf{cs}}\cdot(2\cdot\mathsf{cs}-\ell)^{(2\cdot\mathsf{cs}-\ell)}} \\ &< (\frac{\lambda}{n})^{\lambda}\cdot\frac{1}{n^{\ell-4\cdot\mathsf{cs}}},\ \text{for } \lambda \gg 2\cdot\mathsf{cs}-\ell. \end{aligned}$$

□

Essentially, Theorem 1 and Corollary 1 state that SLiC achieves negligible adversarial success probability in both security parameter λ and number of events ℓ, matching traditional security notions. As shown in the main body of the paper (Section 4.4), adversarial advantage is also low in practice, i.e., for typical real-world parameters of SLiC.

B Optimistic Recover

```
Input: State $\Sigma$ to check, initial state $\Sigma_0$
Output: Recovered log events $\{m_1, \ldots, m_{n'}\}$
// Let $\Sigma_0 = (K_0, seed_0, \mathcal{S}_0)$
// Parse $\Sigma$ as $(K_{n'}, seed_{n'}, \mathcal{S}' = \pi'(s'_1, \ldots, s'_{n' \leq n}))$, let $s'_i = (c'_i, h'_i)$
$\mathcal{R}_{n'-\mathsf{cs}} = \ldots = \mathcal{R}_{n'+\mathsf{cs}} = \varnothing$;
$\mathcal{E}_{n'-\mathsf{cs}} = \ldots = \mathcal{E}_{n'+\mathsf{cs}} = \varnothing$;
// Evolve key, seed, $\pi, \pi^{-1}$
for $i = 1$ to $n' - \mathsf{cs} - 1$ do
  $K_i = \mathsf{PRF}_{K_{i-1}}(\chi)$; $seed_i = \mathsf{PRF}_{seed_{i-1}}(\chi')$;
  Update $\pi$ and $\pi^{-1}$;
end
// Outer Loop
for $i = 1$ to $n' + \mathsf{cs}$ do
  $K_i = \mathsf{PRF}_{K_{i-1}}(\chi)$;
  // Inner Loop
  for $j = n' - \mathsf{cs}$ to $n' + \mathsf{cs}$ do
    $pos \xleftarrow{\$\mathsf{PRG}(seed_i)} \{1, \ldots, j\}$;
    Update $\pi$ and $\pi^{-1}$;
    $\mathcal{E}_j = \mathcal{E}_j \cup \{j, \pi^{-1}[pos]\}$;
    if $\mathsf{HMAC}_{K_i}(c'_{\pi[i]}) = h'_{\pi[i]}$ then
      $\mathcal{R}_j = \mathcal{R}_j \cup \{\mathsf{Dec}_{K_i}(c'_{\pi[i]})\}$;
      break Inner Loop;
    end
  end
end
// Check plausibility
if $\nexists j : [(|\mathcal{R}_j| \geq \lambda - \mathsf{cs}) \wedge (\forall i \in \{1, \ldots, j\} : \{(i, \cdot)\} \in \mathcal{R}_j \oplus i \in \mathcal{E}_j)]$ then
  output $\perp$
else
  output $\mathcal{R}_j$;
end
```

Algorithm 5: Optimistic $\mathsf{Recover}^{\mathsf{Opt}}(\Sigma, \Sigma_0)$