Steady

A Simple End-to-End Secure Logging System

Tobias Pulls and Rasmus Dahlberg

Karlstad University, Dept. of Mathematics and Computer Science, Sweden
first.last@kau.se

Abstract. We present Steady: an end-to-end secure logging system engineered to be simple in terms of design, implementation, and assumptions for real-world use. Steady gets its name from being based on a steady (heart)beat of events from a forward-secure device sent over an untrusted network through untrusted relays to a trusted collector. Properties include optional encryption and compression (with loss of confidentiality but significant gain in goodput), detection of tampering, relays that can function in unidirectional networks (e.g., as part of a data diode), cost-effective use of cloud services for relays, and publicly verifiable proofs of event authenticity. The design is formalized and security proven in the standard model. Our prototype implementation ($\approx 2,200$ loc) shows reliable goodput of over 1M events/s (≈160 MiB/s) for a realistic dataset with commodity hardware for a device on a GigE network using 16 MiB of memory connected to a relay running at Amazon EC2.

Keywords: Secure Logging · Protocols · Applied Cryptography.

1 Introduction

Logs play a vital role during the operational phase of systems by providing a communication channel that gives insights into how systems are operating, such as informing about errors, warnings, or potential security events. Such information have far reaching implications in today's increasingly digitalised world, for example in criminal cases due to so-called Data Retention laws mandating logging for law-enforcement purposes[1] or for general auditing of systems [11]. Due to the increasing number of systems in operation, logs are often transported over a network—potentially stored temporarily by one or more relays—to be collected for centralised analysis that correlate logs, e.g., by using a security information and event management (SIEM) system. The centralised system serves as the primary means of monitoring operations. Absence of logs from a system expected to be operating is a case for concern and typically closely monitored.

Because logs are important they have to be secured and the literature contains a number of contributions on *secure logging*, addressing a wide-range of aspects ranging from schemes for efficient integrity protection to complete systems

[1] For example Directive 2006/24/EC http://europa.eu/!BM68tq, accessed 2018-08-08.

that typically considers log confidentiality in addition to integrity protection. Secure logging *schemes* that do not provide confidentiality typically have to be combined at least with some form of transport security (e.g., TLS) to provide comparable security properties to secure logging *systems*. Further, some secure logging systems that encrypt logs still need other properties from transport security protocols for real-world use, such as replay protection or communication partner authenticity.

In this paper, we present a secure logging system based on several observations made earlier. First, our system is named *Steady* because it relies on a steady (heart)beat of new log events from the generating system for some of its security properties. As mentioned before, monitoring uptime of critical systems is already common. Second, Steady supports untrusted relays for intermediate storage during log transport. This opens up for using public cloud services as relays. Further, for real-world deployment Steady does not rely on any other transport security protocol like TLS for its security properties. Finally, Steady supports efficient publicly verifiable proofs of event authenticity to support use-cases where third-parties need to verify the authenticity of logged events.

Our contributions are:

- The design of a simple secure logging system named Steady that supports untrusted relays and that bases part of its security on the time between blocks of events (Section 2). The system can be used with a data diode in high security settings and is well suited for outsourcing to cloud providers.
- A formal definition of Steady with proofs of security in the standard model for event secrecy, event integrity, delayed event deletion-detection, and unforgeable proofs of event authenticity (Sections 3-4). The security reduces to standard properties of the primitives for hashing, signing, and encrypting.
- A prototype implementation in C and Go instantiated using primitives from libsodium for a 128-bit security level together with a performance evaluation focused on event generation for different relay locations (Section 5). Our evaluation shows reliable goodput of over 1M events/s (≈160 MiB/s) for a realistic dataset with commodity hardware for a device on a GigE network using 16 MiB of memory connected to a relay running at Amazon EC2.

Besides the sections referenced above, related work is presented in Section 6 and Section 7 concludes this paper.

2 Overview of Steady

Figure 1 shows our setting with three different types of systems:

Device a forward-secure system that generates events as part of a log.

Relay an untrusted system that stores events temporarily. A relay *has finite storage*, so events must be dropped due to space constraints.

Collector a trusted system that collects and verifies events. We assume that a collector has sufficient space and processing power available.

Without loss of generality we consider a single relay but stress that Steady supports multiple relays by cascading writes (as shown later).

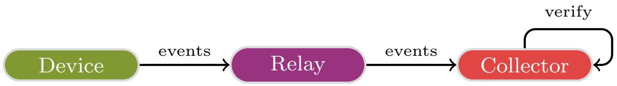


Fig. 1: The setting of Steady with device, relay, and collector systems.

2.1 Threat Model

Our ultimate goal is end-to-end security from device to collector while considering all relays and the network as active adversaries during logging. The collector is assumed to be trusted. Further, we consider the device forward secure: it is initially trusted from setup up until a point in time t when an adversary compromises the device. We only aim to secure events generated a delta δ time *prior to compromise*: this is because we base some security properties of Steady on heartbeats of new blocks from the device. For example, if $t = 120$ seconds since setup and $\delta = 10$ seconds, then events generated before $t - \delta = 110$ seconds after setup are fully protected. We stress that δ is user-configurable and a practical trade-off (discussed later in Section 4) that enables us to make Steady simple.

2.2 Properties and Requirements

Informally, events should not be possible to tamper with and optionally also confidential (encrypted). It should be possible to buffer events at the device and optionally compress them, despite leaking information[2]. Compression and encryption are optional because in some settings encryption may not be needed—e.g., if traffic is wrapped in some other secure transport protocol—and compression typically gives a significant throughput improvement. Regardless of encryption or compression, it should be possible to produce an independent proof of each event that be used to convince a third party of the authenticity of the event.

Beyond being untrusted, relays should be able to have fixed storage for sake of operational concerns (e.g., to fit all relayed data in memory) and to be able to optimally use Cloud Service Providers like Amazon EC2. After setup we want no direct communication between device and collector: this ensures that a relay can be part of a unidirectional network for high-security (air-gapped) settings.

2.3 Setup and Policy Creation

Figure 2 shows the setup of Steady. It starts with the collector generating a public key (pub) that is sent to the device together with a *timeout* value and the *minimum storage space* for its relay. The timeout specified the maximum amount of time between events from the device, described further later. The device commits to the parameters by creating a signed *policy* with a key-pair that has been verified to belong to the device out of band. A policy is valid

[2] Compression breaks semantic security and depending on setting completely neglects any encryption [7], as shown, e.g, in the CRIME and BREACH attacks.

for the lifetime of logging. The policy is propagated to the untrusted relay. The relay verifies that the storage parameter in the policy is acceptable and that the signature is valid. Finally, the collector polls the relay for the policy and verifies it. Each entity has its respective state (defined later), where notably the private and signing keys are only known to the respective entities that generated them.

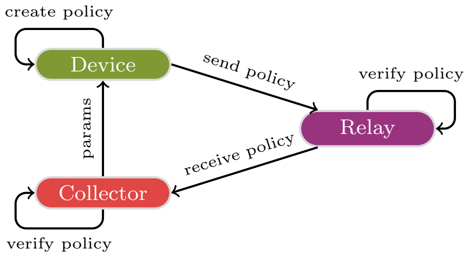


Fig. 2: The setup of Steady resulting in a signed policy of parameters.

2.4 Device Logging and Creating Blocks

The device generates a *block* of events periodically, at least when the timeout triggers. Blocks are given an incremental *index* by the device together with other metadata such as a timestamp, a signature, and the root of a Merkle tree over all events [10]. A more precise definition is given later. Events are kept in a queue as shown in Figure 3 before being included in a block. Several events in the same block makes compression more efficient and amortises costs related to cryptographic operations during block generation and network transport.

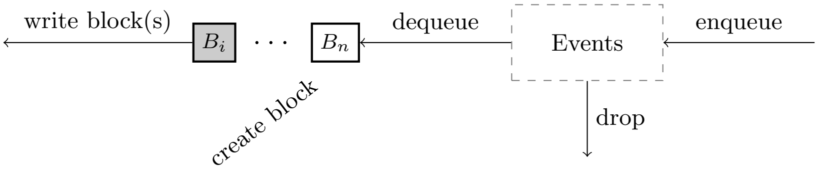


Fig. 3: The device's event and block generation flow.

Note that if a device has to drop events, e.g., due to memory constraints, then they should be dropped from the queue and not from potentially buffered blocks not yet written to the relay. While it is possible for the device to recreate blocks not yet written to the relay, it is relatively costly to do so. Any metadata to report to the collector about dropped events can be sent as part of an event.

2.5 Writing to and Reading From a Relay

When a device *writes* one or more blocks to the relay, the relay first sorts the blocks based on *index*, and then processes one block at the time as shown in Figure 4. The relay verifies the signature on the block and only accepts if the block has been signed with the same key as the policy. Then it ensures that the block is the next (in terms of index) block based on the previous block and if so makes space for the block before accepting (storing) it. To keep at most the minimum storage space in the policy, the relay stores a (FIFO) queue of blocks.

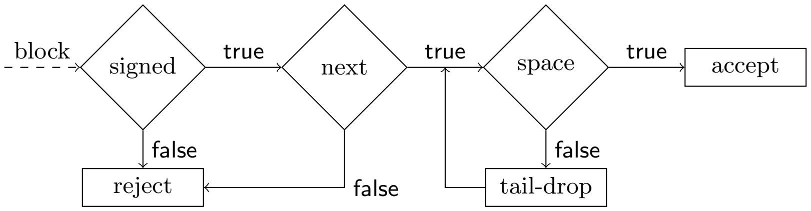
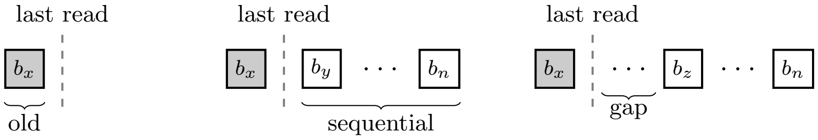


Fig. 4: The relay's event write flow.

A *read* from the relay is done based on a supplied block index by traversing the queue and sending each block with an equal or greater index. Because a relay is defined as a FIFO queue with read and write operations, multiple relays can be used where writes cascade and the collector reads from the last relay.

2.6 Collector Verification

The collector periodically polls for new blocks from the relay based on the latest read block index in its state. Because each block is signed we know that events within it cannot be tampered with, so verification focuses on ensuring we get the expected blocks. Figure 5 shows the three possible *correct* cases when reading from a relay (with read index y): (a) no new blocks, (b) a sequence of new blocks directly following the last read block ($x+1 = y$), or (c) a disconnected sequence.



(a) Empty (b) Connected sequence (c) Disconnected sequence

Fig. 5: The three possible *correct* cases when reading blocks.

Based on the above three cases we sketch a verification algorithm verify that verifies that we read the expected blocks from the relay:

Empty Valid if the time since the last block is less then the timeout.

Connected sequence Valid if the timestamp of the most recent returned block is timely given the current time at the collector and the timeout.

Disconnected sequence Same as the connected sequence case with the additional requirement that the size of all returned blocks is consistent with the minimum storage space in the policy.

The complete formally defined algorithm is available in Appendix A.

2.7 Proof Generation and Verification for Event Authenticity

As mentioned in Section 2.4, each block is signed by the device and contains the root of a Merkle tree over all events in the block. A proof of event authenticity is simply a Merkle audit path to an event in question and metadata from the block to verify the signed root from the device. Verifying the proof involves verifying the signature and comparing the computed Merkle root from the audit path to the one signed by the device.

3 Formal Model of Steady

We formally model Steady, starting with core definitions and the logging scheme in Section 3.1 followed by properties in Section 3.2.

3.1 Core Definitions and the Logging Scheme

We define a policy (Definition 1), block (Definition 2), and proof (Definition 3).

Definition 1 (Policy). *Given a public key* pub*, a signing key-pair* $(\mathsf{sk}, \mathsf{vk})$*, a timeout* t*, a minimum space* s*, the current time* τ*, and a policy identifier* k*, a policy* P *is defined as:*

$$\{k, \mathsf{vk}, \mathsf{pub}, t, s, \tau, \sigma\}$$

where σ *is a signature using* sk *over all other values.*

Nothing prevents the relay from storing more blocks than the minimum space s. The policy identifier k should be unique per device. The timeout t is the maximum period of time (inclusive) between blocks being created by the device. The current time τ is included as relevant metadata and a reference for when the first block should be expected the latest (relative to t).

Definition 2 (Block). *Given a policy identifier* k*, a signing key* sk*, a list of events* $\boldsymbol{e}$ *with the Merkle tree root* r*, an optionally compressed and then encrypted payload* $p \leftarrow \boldsymbol{e} \| \mathsf{IV}$*, a block index* i*, two boolean flags indicating encryption* f_e *and* *compression f_c, the size of the previous block ℓ_p, an initialization vector* IV*, and the current time τ, a block B is defined as:*

$$\big\{i, \ell_c, \ell_p, \mathsf{H}_k(p), \phi \leftarrow \mathsf{H}_k\big(i\|\ell_c\|\ell_p\|\mathsf{H}_k(p)\|f_e\|f_c\big), \iota \leftarrow \mathsf{H}_k(r\|\mathsf{IV}), \tau, \sigma, p\big\}$$

where $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, \phi\|\iota\|\tau)$ and ℓ_c is the resulting size of the current block. We refer to ϕ as the block header hash and ι the root hash.

Note that H_k is a keyed hash function, where the key puts the hashes of different blocks belonging to different policies into different domains. We need the previous block size ℓ_p in the verification algorithm, see Appendix A. Based on a chosen or guessed $\boldsymbol{e}$, an adversary can reconstruct the Merkle tree root r' and check if it matches $r \in B$. Since this would violate event secrecy (defined later), a block includes $\mathsf{H}(r\|\mathsf{IV})$ instead of r itself. IV is part of the encrypted payload. Note that the signature does not directly cover p, and that the block sans the payload is constant size. This enables a block to be efficiently streamed, such that the verifier can verify the signature using only the block header and then know how large the remainder of the block should be (using ℓ_c). The hash of the payload ($\mathsf{H}(p)$) in the header authenticates it once read. Finally, the structure ensures compact proofs of event authenticity that does not leak block metadata[3].

Definition 3 (Proof of event authenticity). *Given a block B and an event $e \in B$, a proof Π is defined as:*

$$\left\{\mathsf{IV}, \phi, \tau, \sigma, \overrightarrow{h}^{e}\right\}$$

where $\sigma \in B$ and $\overrightarrow{h}^{e}$ is a Merkle audit path to e that enables the computation of ι using IV*, which in turn is signed by σ together with τ and ϕ.*

Having defined a policy, block, and proof we can now define a Steady logging scheme (Definition 4):

Definition 4 (Steady scheme). *Given a security parameter λ, a time-skew security parameter δ, an encryption key-pair* $(\mathsf{priv}, \mathsf{pub})$*, a signing key-pair* $(\mathsf{sk}, \mathsf{vk})$*, and a policy P, we define a collector state S_c as $\{\delta, \mathsf{priv}, P, i, \tau\}$, a device state S_d as $\{\mathsf{pub}, \mathsf{sk}, P, i, \tau, \ell_p\}$, and a relay state S_r as $\{P, \mathcal{B}\}$. Here i is the (expected) next block index, τ the time of the most recent block, ℓ_p the size of the previous block, and $\mathcal{B}$ a sequence of blocks; initially i is set to 0, ℓ_p to 0, $\mathcal{B}$ to an empty sequence, and τ to the creation time of P. A Steady scheme $\mathcal{S}$ is composed of seven algorithms* $\{\mathsf{setup}, \mathsf{block}, \mathsf{read}, \mathsf{write}, \mathsf{verify}, \mathsf{proofGen}, \mathsf{proofVer}\}$ *that are polynomial in space and time:*

- $S_\mathsf{c}, S_\mathsf{d}, S_\mathsf{r} \leftarrow \mathsf{setup}(\lambda, \delta, t, \tau, s)$*: given two security parameters λ and δ, a timeout $t \in \mathbb{N}$, the current time τ, and a space $s \in \mathbb{N}$,* setup *runs* $(\mathsf{priv}, \mathsf{pub}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)$ *and* $(\mathsf{sk}, \mathsf{vk}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)$*. Next, a policy P is generated from $k \leftarrow_\$ 1^\lambda$,* pub*,* $(\mathsf{sk}, \mathsf{vk})$*, t, τ, and s. The output is the initial states S_c, S_d, S_r.*

[3] The block metadata i, ℓ_c, and ℓ_p are hashed together with the hash of the payload that is likely high entropy, unlike the metadata.

- $B, S'_{\mathsf{d}} \leftarrow \mathsf{block}(S_{\mathsf{d}}, \boldsymbol{e}, \tau, f_e, f_c)$*: given a device state* S_{d}*, a list of events* $\boldsymbol{e}$*, the current time* τ*, an encryption flag* f_e*, and a compression flag* f_c*,* block *generates the next block* B *based on* S_{d}*,* τ*, and the flags. The output is* B *and a refreshed state* S'_{d} *such that* $i' = i + 1, \ell'_p = \ell_c$*, and* $\tau' = \tau$*.*
- $\beta \leftarrow \mathsf{read}(S_{\mathsf{r}}, i)$*: given a relay state* $\{P, \mathcal{B}\} \leftarrow S_{\mathsf{r}}$ *and a block index* i*,* read *outputs a sequence of blocks* $\beta \leftarrow \{B_j \mid B_j \in \mathcal{B} \wedge j \geq i\}$*.*
- $S'_{\mathsf{r}} \leftarrow \mathsf{write}(S_{\mathsf{r}}, \beta)$*: given a relay state* $\{P, \mathcal{B}\} \leftarrow S_{\mathsf{r}}$ *and a connected sequence of blocks* β*,* write *outputs a refreshed state* S'_{r} *with* $\mathcal{B}' \subseteq \mathcal{B} \cup \beta$*.* $\mathcal{B}'$ *should contain at least as many of the most recent blocks as implied by the minimum space parameter in* P*.*
- $\alpha, S'_{\mathsf{c}} \leftarrow \mathsf{verify}(S_{\mathsf{c}}, \mathcal{B}, \tau)$*: given a collector state* S_{c}*, a sequence of blocks* $\mathcal{B}$*, and the current time* τ*,* verify *determines with respect to* S_{c} *and* τ *if every* $B \in \mathcal{B}$ *is a valid block and that no block is missing. The output is an answer* $\alpha \leftarrow \{\mathsf{false}, \mathsf{true}\}$ *together with a refreshed state* S'_{c} *that matches* $i+1$ *and* τ *from the most recent valid block.*
- $\Pi \leftarrow \mathsf{proofGen}(B, e, S_{\mathsf{c}})$*: given a block* B*, an event* $e \in B$ *and collector state* S_{c} *used to retrieve and verify* B*,* proofGen *outputs a membership proof* Π*.*
- $\alpha \leftarrow \mathsf{proofVer}(\Pi, e, \mathsf{vk})$*: given a proof* Π*, an event* e*, and a verification key* vk*,* proofVer *outputs an answer* $\alpha \leftarrow \{\mathsf{false}, \mathsf{true}\}$ *that is* true *iff* Π *shows that* vk *authenticates* e*, otherwise* false*.*

3.2 Properties

For the following correctness and security definitions, we define a helper algorithm $\{\mathsf{false}, \mathsf{true}\} \leftarrow \mathsf{check}(S_{\mathsf{r}}, \tau)$ that outputs true iff the state S_{r} is without deletions that violate the policy $P \in S_{\mathsf{r}}$ given the (correct) current time τ.

Definition 5 (Correctness). *Let* $\mathcal{S}$ *be an instance of the Steady logging scheme* $\{\mathsf{setup}, \mathsf{block}, \mathsf{read}, \mathsf{write}, \mathsf{verify}, \mathsf{proofGen}, \mathsf{proofVer}\}$*. For all* $\lambda, \delta, s, t \in \mathbb{N}$*, policy creation time* τ*, and* $S_{\mathsf{c}}, S_{\mathsf{d}}, S_{\mathsf{r}} \leftarrow \mathsf{setup}(\lambda, \delta, t, \tau, s)$ *that are followed by polynomially many invocations of* block*,* write*, and* verify *to obtain a sequence of blocks* $\mathcal{B}$ *and the intermediate states* $S_{\mathsf{c}}, S_{\mathsf{d}}, S_{\mathsf{r}}$*, then* $\mathcal{S}$ *is correct iff:*

$$Pr\begin{bmatrix} \forall e \in \forall \mathcal{B} \leftarrow \mathsf{read}(S_{\mathsf{r}}, i \in S_{\mathsf{c}}) : \mathsf{check}(S_{\mathsf{r}}, \tau) \wedge \neg \mathsf{verify}(S_{\mathsf{c}}, \mathcal{B}, \tau) \\ \vee \neg \mathsf{proofVer}\big(\Pi \leftarrow \mathsf{proofGen}(B, e, S_{\mathsf{c}}), e, \mathsf{vk} \in S_{\mathsf{c}}\big) \end{bmatrix} \leq \mathsf{negl}(\lambda)$$

In the following security definitions, we consider an adversary that controls an instance of the Steady scheme adaptively. Whereas access to S_{r} is unlimited with write, the states S_{c} and S_{d} can only be influenced through oracle access:

- $B \leftarrow \mathsf{oblock}(\boldsymbol{e}, f_e, f_c)$: given a list of events $\boldsymbol{e}$, encryption flag f_e, and compression flag f_c, oblock runs block with the provided input, the most recent device state S_{d} (kept by the oracle), and the correct current time. Returns the generated block B.
- $\alpha \leftarrow \mathsf{overify}(\mathcal{B})$: given a sequence of blocks $\mathcal{B}$, overify runs verify with $\mathcal{B}$, the most recent collector state S_{c}, and the current time τ. Returns the answer α.

- $\boldsymbol{e}, \mathsf{IV} \leftarrow \mathsf{odec}(B)$: given a block B with an encrypted payload, odec decrypts the payload in B and returns the list of events $\boldsymbol{e}$ and IV.

For event secrecy (Definition 6), an oracle $B \leftarrow \mathsf{oblock}{*}_b(\boldsymbol{e}_0, \boldsymbol{e}_1)$ is defined that outputs the next block B from $\boldsymbol{e}_b$ using oblock with $f_e = \mathsf{true}$, where b is a secret challenge bit. In order to prevent size correlations, $\mathsf{oblock}{*}_b$ only accepts $\boldsymbol{e}_i$ of equally many events that match pair-wise in size and $f_c = \mathsf{false}$. The adversary may not use odec to decrypt any block from $\mathsf{oblock}{*}_b$ (as in IND-CPA).

Definition 6 (Event secrecy). *For all $\lambda, \delta \in \mathbb{N}$ and PPT adversaries $\mathcal{A}$, a Steady scheme provides computational secrecy of an event's content iff:*

$$\frac{1}{2}\left|\Pr\left[\mathsf{Exp}^{es}_{\mathcal{A}}(\lambda,\delta)=1 \mid b=0\right]+\Pr\left[\mathsf{Exp}^{es}_{\mathcal{A}}(\lambda,\delta)=1 \mid b=1\right]-1\right| \le \mathsf{negl}(\lambda)$$

```
$\mathsf{Exp}^{es}_{\mathcal{A}}(\lambda,\delta)$:
$t, \tau, s, S_\mathsf{a} \leftarrow \mathcal{A}(\lambda,\delta)$
$S_\mathsf{c}, S_\mathsf{d}, S_\mathsf{r} \leftarrow \mathsf{setup}(\lambda,\delta,t,\tau,s), b \leftarrow_{\$} \{0,1\}$
$b' \leftarrow \mathcal{A}^{\mathsf{oblock},\mathsf{oblock}*_b,\mathsf{read},\mathsf{write},\mathsf{overify},\mathsf{odec}}(S_\mathsf{r}, S_\mathsf{a})$
return $b \overset{?}{=} b'$
```

For event integrity (Definition 7), we define an algorithm $\{\mathsf{false}, \mathsf{true}\} \leftarrow \mathsf{valid}(P, B, e, e')$ that uses verify as a subroutine. The output is true iff B is a block for the policy P when $e \in B$ and after replacing the event $e \in B$ with e'.

Definition 7 (Event integrity). *For all $\lambda, \delta \in \mathbb{N}$ and PPT adversaries $\mathcal{A}$, a Steady scheme provides integrity of an event's content iff:*

$$\Pr\left[\mathsf{Exp}^{ei}_{\mathcal{A}}(\lambda,\delta)=1\right] \le \mathsf{negl}(\lambda)$$

```
$\mathsf{Exp}^{ei}_{\mathcal{A}}(\lambda,\delta)$:
$f_e, f_c, t, \tau, s, S_\mathsf{a} \leftarrow \mathcal{A}(\lambda,\delta)$
$S_\mathsf{c}, S_\mathsf{d}, S_\mathsf{r} \leftarrow \mathsf{setup}(\lambda,\delta,t,\tau,s)$
$B, e, e' \leftarrow \mathcal{A}^{\mathsf{oblock},\mathsf{read},\mathsf{write},\mathsf{overify},\mathsf{odec}}(S_\mathsf{r}, S_\mathsf{a})$
return $e \neq e' \wedge \mathsf{valid}(P \in S_\mathsf{c}, B, e, e')$
```

For delayed event deletion detection (Definition 8), the adversary gets control over the setup parameters except for the current time τ. After polynomial invocations to the listed oracles, the adversary outputs a state of the relay S_r, a positive time duration Δ that represents the expired time after setup, and the duration x since τ when the latest (in terms of index) block was modified or deleted in S_r. The adversary wins if she can modify or delete with more than negligible probability one or more blocks from S_r without detection by verify after the duration $t+\delta$, where t is the timeout and δ a security parameter. Note that $\mathsf{overify}$ is not available to the adversary: we remove this capability to ensure that the call to read using S_c reads all blocks in S_r for verify to verify[4].

[4] If the adversary can modify or remove a block already read from the relay by the collector this would cause check to fail but this is not relevant for security.

Definition 8 (Delayed event deletion detection). *For all $\lambda, \delta \in \mathbb{N}$ and* PPT *adversaries $\mathcal{A}$, a Steady scheme provides delayed event deletion detection iff:*

$$\Pr\left[\mathsf{Exp}^{dedd}_{\mathcal{A}}(\lambda,\delta) = 1\right] \leq \mathsf{negl}(\lambda)$$

```
$\mathsf{Exp}^{dedd}_{\mathcal{A}}(\lambda,\delta)$:
$f_e, f_c, t, s, S_\mathsf{a} \leftarrow \mathcal{A}(\lambda,\delta)$
$S_\mathsf{c}, S_\mathsf{d}, S_\mathsf{r} \leftarrow \mathsf{setup}(\lambda,\delta,t,\tau,s)$, where $\tau$ is the correct current time
$S_\mathsf{r}, \Delta, x \leftarrow \mathcal{A}^{\mathsf{oblock},\mathsf{read},\mathsf{write},\mathsf{odec}}(S_\mathsf{r}, S_\mathsf{a})$
return $\neg\mathsf{check}(S_\mathsf{r}, \tau+\Delta) \wedge \mathsf{verify}\Big(S_\mathsf{c}, \mathsf{read}(S_\mathsf{r}, i \in S_\mathsf{c}), \tau+\Delta\Big) \wedge x > t+\delta$
```

For unforgeable proofs of event authenticity (Definition 9), the adversary has to output an event e and a valid proof Π for e, where e has not been part of any of the blocks $\mathcal{B}$ generated by the adversary as part of calls to oracle oblock.

Definition 9 (Unforgeable proofs of event authenticity). *For all $\lambda, \delta \in \mathbb{N}$ and* PPT *adversaries $\mathcal{A}$, a Steady scheme provides unforgeable proofs of event authenticity iff:*

$$\Pr\left[\mathsf{Exp}^{ufp}_{\mathcal{A}}(\lambda,\delta) = 1\right] \leq \mathsf{negl}(\lambda)$$

```
$\mathsf{Exp}^{ufp}_{\mathcal{A}}(\lambda,\delta)$:
$f_e, f_c, t, \tau, s, S_\mathsf{a} \leftarrow \mathcal{A}(\lambda,\delta)$
$S_\mathsf{c}, S_\mathsf{d}, S_\mathsf{r} \leftarrow \mathsf{setup}(\lambda,\delta,t,\tau,s)$
$\mathcal{B}, e, \Pi \leftarrow \mathcal{A}^{\mathsf{oblock},\mathsf{read},\mathsf{write},\mathsf{overify},\mathsf{odec}}(S_\mathsf{r}, S_\mathsf{a})$
return $e \notin \mathcal{B} \wedge \mathsf{proofVer}(\Pi, e, \mathsf{vk} \in P \in S_\mathsf{c})$
```

4 Security of Steady

4.1 Assumptions

Lemma 1. *In a Merkle tree, the security of an audit path reduces to the collision resistance of the underlying hash function* H*.*

Proof (sketch). This follows from the work of Merkle [10] and Blum *et al.* [1].

4.2 Properties and Proofs

Note that the full verify algorithm is in the Appendix A.

Theorem 1 (Correctness). *Steady is correct as in Definition 5.*

Proof (sketch). For the first part of Definition 5, regarding verification, all possible (valid) blocks are per definition generated by calls to block, written to the relay with write, and returned by read from a valid S_r given time τ. This implies that blocks are *timely*, form a valid *sequence*, and the size is at least the *size in the policy*, so verify accepts with probability 1 for all possible blocks. For the second part of Definition 5, regarding proofs, this follows trivially from the definitions of proofGen and proofVer.

Theorem 2 (Event secrecy). *For an IND-CPA secure public-key encryption scheme and a pre-image resistant hash function, Steady provides computational secrecy of events as in Definition 6.*

Proof (sketch). The events have been encrypted together with a uniformly random IV using an IND-CPA secure public-key encryption scheme. This ensures that the payload itself is not a distinguisher. Further, as part of the block B from oblock$*$, (i, ℓ_p, τ) are independent of the events in the block. The root hash ι is done with a pre-image resistant hash function, where the root of the Merkle tree is combined with a uniformly random IV before being hashed. The payload hash, block header ϕ, and signature σ operate on the encrypted events. □

Event secrecy, per definition, is only provided when encryption is enabled and compression is disabled. Further, because of the use of public-key encryption, Steady also provides forward secrecy upon compromise of a device and its state.

Corollary 1 (Forward event secrecy). *For an IND-CPA secure public-key encryption scheme and a pre-image resistant hash function, Steady provides computational forward secrecy of events in blocks.*

Theorem 3 (Event Integrity). *Given an existentially unforgeable signature scheme and a collision resistant hash function, Steady provides computational integrity of events as defined in Definition 7.*

Proof (sketch). An existentially unforgeable signature scheme prohibits forgery of different messages. This means that every signed block B will include an authentic Merkle tree root that cannot be replaced by the adversary, and each event $e \in B$ will thus be fixed by an audit path in the block's Merkle tree. The security of an audit path reduces to the collision resistance of the underlying hash function (Lemma 1). □

Theorem 4 (Unforgeable Proofs of Event Authenticity). *Given an existentially unforgeable signature scheme and a collision resistant hash function, Steady's proofs of event authenticity are unforgeable as defined in Definition 9.*

Proof (sketch). Proof verification consists of verifying an audit path in a Merkle tree and that the root of the Merkle tree is signed. The security of an audit path reduces to the collision resistance of the underlying hash function (Lemma 1) and the signature is existentially unforgeable. □

Theorem 5 (Delayed event deletion detection). *Given an existentially unforgeable signature scheme and a collision resistant hash function, Steady provides delayed event deletion detection as defined in Definition 8.*

Proof (sketch). Within blocks, events cannot be deleted because each event is fixed by an audit path in a Merkle tree (Lemma 1) where the root is signed with an existentially unforgeable signature. Therefore we need to show that blocks cannot be deleted without detection.

With a correct collector state S_c generated through setup or oracle calls to $\mathsf{overify}$, read returns a sequence of blocks $\mathcal{B}$. The authenticity and integrity of blocks are protected by an existentially unforgeable signature. The verification algorithm verify sorts $\mathcal{B}$ with valid signatures into a (not necessarily connected) sequence $(B_0 \dots B_n)$ based on block index i, and removes any duplicates or old blocks (index $<$ requested index to read). There are then three *possible* cases:

Case 1 $\mathsf{len}\big((B_0 \dots B_n)\big) = 0$: read returned no new blocks.

Case 2 $\mathsf{len}\big((B_0 \dots B_n)\big) > 0 \wedge B_0^i = S_\mathsf{c}^i$: read provided one or more new blocks and the first block is the block directly after the previous read.

Case 3 $\mathsf{len}\big((B_0 \dots B_n)\big) > 0 \wedge B_0^i \neq S_\mathsf{c}^i$: read provided one or more new blocks and the first block is *not* the block directly after the previous read.

Note that for $\mathsf{Exp}_{\mathcal{A}}^{\text{dedd}}(\lambda, \delta)$ to return true then $x > t+\delta$, meaning that at least $t+\delta$ time must have expired since the latest block was deleted or modified by the adversary. Further, the policy specifies that (i) a block should be produced at least after t time since the latest block, and (ii) the relay should store at least s space of the most recent blocks. If a block has been deleted from S_r such that check returns false, then the policy has been violated at time $\tau + \varDelta$ (per definition). Therefore either of the two, or both, parts of the policy have been violated. Returning to the three possible cases in the paragraph above:

Case 1 The timely check will detect any deleted block because (i) $t+\delta$ time has expired, (ii) the time in the collector's state is based on an existentially unforgeable signature in the last verified block, and (iii) the provided current time $\tau + \varDelta$ is the same as for check.

Case 2 In addition to a timely check—but in this case based on the time in the latest new block instead of state—verify also checks that all blocks form a connected sequence from the block in state to the latest block. All blocks are signed with an existentially unforgeable signature.

Case 3 In addition to a timely check of the latest new block, verify checks that all new blocks form a connected sequence, and that the size of the new blocks together with the size of the prior block ($B_0^{\ell_p}$) is greater than the space s in the policy, detecting any deleted blocks.

□

Theorem 5 covers block deletion as defined in Definition 8 when reading blocks from the relay that the adversary completely controls. Further, per definition write checks for a monotonically increasing block index, therefore replay attacks are irrelevant *after setup* (Corollary 2).

Corollary 2 (Relay replay attacks). *Given an existentially unforgeable signature scheme, writes and reads in Steady are secure from replay attacks.*

4.3 Relay Flushing and Device Forward Security

Note that the delayed event deletion detection, as defined in Definition 8, is limited in several ways. First, deletion detection is delayed by the timeout t and time-skew δ parameters. Benign delays due to, e.g., network effects or clock drift between device and collector, risks causing *false positives* with a time-based deletion detection mechanism. We therefore introduce a security parameter δ that specifies the acceptable delay for the collector, trading *delayed* deletion detection for reduced false positives.

Further, because the relay only is required to keep finite storage, this opens up another venue for an adversary to "flush" a relay (Corollary 3). We stress that this is fundamental restriction in the setting.

Corollary 3 (Relay flushing). *An adversary with the capability to trigger the device to create new blocks can flush blocks from the relay that have yet to be read by the collector. Accordingly, a relay's minimum storage capacity and the collector's reading frequency must be treated as security parameters.*

In the setting of finite storage relays, forward event integrity and forward (delayed) event deletion detection are less relevant: if the collector does not read (fast enough) an adversary can flush the relay, and if a device blocks or discards new events when storage is full (or not read) then this is either a severe denial-of-service vector or just a precondition for an adversary to trigger before launching an attack she does not want detected (the same outcome as being able to compromise and delete events).

Corollary 4 (Collector reads and device forward security). *If the collector continuously reads from the relay, then the timeout and time-skew parameters give an upper bound for the time the adversary has in undetectably modifying or deleting events that have yet been read by the collector.*

5 Performance Evaluation

We first instanciated our scheme to reach a 128-bit security level with BLAKE2b-256, AES256-GCM, X25519, and Ed25519 using an NaCl box-like scheme for encryption[5]. The device is implemented[6] in C (c11) in 987 loc (as reported by `cloc`) using libsodium[7] for crypto primitives and LZ4[8] for compression. The relay and a minimal collector are implemented[9] in Go in 1239 loc.

[5] NaCl box (https://nacl.cr.yp.to/box.html) uses Salsa20 and Poly1305, we use AES256-GCM instead for the hardware speedup on selected platforms.

[6] https://github.com/pylls/steady-c, Apache 2.0 license.

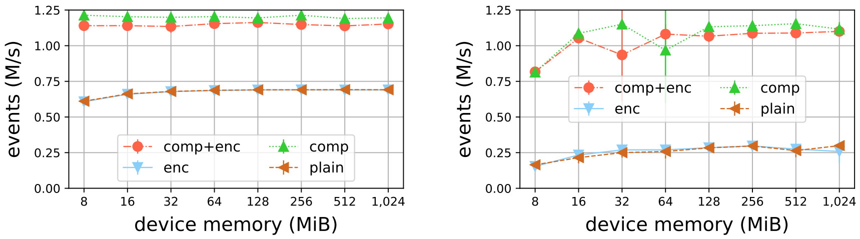
[7] https://libsodium.org/, accessed 2018-08-05.

[8] https://lz4.github.io/lz4/, accessed 2018-08-05.

[9] https://github.com/pylls/steady, Apache 2.0 license.

For our performance evaluation focused on event goodput we used 1 GiB of syslog events (6,472,046 events, mean size 164.9±39.7 bytes) of the Dartmouth campus CRAWDAD dataset [8]. The device was run on an i7-4790@3.6GHz CPU with 16 GiB DDR3@1600MHz memory and a 1 Gib/s Internet connection. It was limited to using one (logical) core for block creation. We hosted the relay at two locations: on a laptop connected throguh a 1 Gib/s LAN (mean 0.4±0.2 ms latency) and at an Amazon EC2 instance type m5.large in Frankfurt with a up to 10 Gib/s connection (mean 29.8±0.2 ms latency). The relay is never CPU bound due to little computation needed to verify blocks (see Figure 4).

Figure 6 shows the results of our performance evaluation. Compression enables a device with only 16 MiB of memory to sustain over 1M events/s (over 160MiB/s) regardless of relay location or use of encryption. Without compression the increased latency to the relay has a significant impact on goodput, likely because the connection between the device and relay is saturated. Device memory beyond 16 MiB has little or no impact on goodput.



(a) Gigabit LAN.

(b) Amazon EC2 m5.large in Frankfurt.

Fig. 6: Events per second as a function of device memory for two relay locations.

6 Related Work

Reasoning about event membership in logs and consistency of logs can use (evolving) Merkle trees [10], as done by Crosby and Wallach [3]. Publicly verifiable schemes (as defined by Holt [5])—that enables membership verification of an event in a log with only public information—typically use signatures on events. Schneier and Kelsey proposed a forward-secure logging system that protects the integrity and confidentiality of events on a per-event basis using MACs and encryption [11]. Grouping events is an important part of providing protection against *truncation* attacks (deletion detection), e.g., as done by Ma and Tsudik [9]. Forward-secure sequential aggregate signatures—introduced in the context of secure logging by Ma and Tsudik [9] and built upon by a number of works, e.g., [14,15]—aggregates signatures over sequential messages into one compact signature instead of individual signatures per message to save storage space and bandwidth. Hartung et al. proposed a provably-secure logging scheme that is also fault-tolerant: the scheme can tolerate a number of manipulated log entries (determined a priori) without invalidating the signature [4]. Steady uses one signature per block over the root of a Merkle tree: more efficient in terms of bandwidth and operations than one signature per message, but more fragile.

Notably, the secure logging system PillarBox by Bowers *et al.* [2] is both complementary and related to Steady in several ways. Both PillarBox and Steady buffers events before transmitting them. The verification algorithm used by PillarBox uses a "gap-checker" to look for missing events, similar to Steady's approach to looking for blocks (Figure 5). As complementary, PillarBox focuses on device compromise, providing integrity of all events buffered prior to compromise and optionally also provides "stealth": hiding the existence of events generated prior to compromise in the buffer. They report event generation in the order of 100,000 events/s (on older hardware). Unlike PillarBox, Steady supports relays, has optional compression, and publicly verifiable proofs of even authenticity.

There are a number of logging systems that use trusted hardware—such as TPM, Intel SGX, and GlobalPlatform TEE—as a basis for system security, also on intermediate systems like our relays [6,12,13]. Steady is software-based.

7 Conclusions

We presented Steady, a simple secure logging system that supports intermediate storage on untrusted relays. Steady is formalised and security proven in the standard model based on vanilla cryptographic primitives and assumptions. The goal of our work was to construct a practical logging system, that does not require other security protocols (e.g., TLS for transport) and that would be reasonably easy to implement and audit. Our performance evaluation which uses a ≈2,200 loc implementation shows significant goodput on resource-constrained devices when the relay is hosted at a popular commercial cloud provider, especially if compression is used. While compression may leak information despite the use of encryption it can be arguably a worthwhile trade-off in many settings.

Acknowledgments. We would like to thank Christian Gotare, Anders Lidén, Mattias Nordlund, and Roel Peeters for valuable feedback. This research as part of the HITS research profile was funded by the Swedish Knowledge Foundation.

References

1. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica **12**(2/3), 225–244 (1994)
2. Bowers, K.D., Hart, C., Juels, A., Triandopoulos, N.: PillarBox: Combating next-generation malware with fast forward-secure logging. In: RAID (2014)
3. Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper-evident logging. In: Monrose, F. (ed.) 18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings. pp. 317–334. USENIX Association (2009)

4. Hartung, G., Kaidel, B., Koch, A., Koch, J., Hartmann, D.: Practical and robust secure logging from fault-tolerant sequential aggregate signatures. In: ProvSec 2017

5. Holt, J.E.: Logcrypt: forward security and public verification for secure audit logs. In: The proceedings of AusGrid and AISW 2006

6. Karande, V., Bauman, E., Lin, Z., Khan, L.: SGX-Log: Securing system logs with SGX. In: AsiaCCS 2017

7. Kelsey, J.: Compression and information leakage of plaintext. In: FSE 2002

8. Kotz, D., Henderson, T., Abyzov, I., Yeo, J.: CRAWDAD dataset dartmouth/campus (v. 2009-09-09). Downloaded from https://crawdad.org/dartmouth/campus/20090909 (Sep 2009)

9. Ma, D., Tsudik, G.: A new approach to secure logging. TOS **5**(1), 2:1–2:21 (2009)

10. Merkle, R.C.: A digital signature based on a conventional encryption function. In: CRYPTO 1987

11. Schneier, B., Kelsey, J.: Cryptographic Support for Secure Logs on Untrusted Machines. In: USENIX Security Symposium. pp. 53–62. USENIX (1998)

12. Shepherd, C., Akram, R.N., Markantonakis, K.: EmLog: Tamper-resistant system logging for constrained devices with TEEs. In: WISTP 2017

13. Sinha, A., Jia, L., England, P., Lorch, J.R.: Continuous tamper-proof logging using TPM 2.0. In: TRUST 2014

14. Yavuz, A.A., Ning, P.: BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems. In: ACSAC 2009

15. Yavuz, A.A., Ning, P., Reiter, M.K.: Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In: FC 2012

A Appendix: Collector Verification Algorithm

The below complete algorithm uses the notation introduced in Section 3.

```
$\mathsf{verify}(S_\mathsf{c}, \mathcal{B}, \tau)$
$(B_0 \ldots B_n) \leftarrow \mathsf{sort}(\mathcal{B})$ // $B_0^i \geq S_\mathsf{c}^i$, signatures valid, duplicate blocks removed
$\alpha \leftarrow \mathsf{validate}\big((B_0 \ldots B_n), \tau, S_\mathsf{c}^\tau, t \in P, s \in P, S_\mathsf{c}^i\big)$
**if** $\alpha \wedge \mathsf{len}\big((B_0 \ldots B_n)\big) > 0$ **then** $S_\mathsf{c}^i \leftarrow B_n^i + 1, S_\mathsf{c}^t \leftarrow B_n^\tau$
**return** $\alpha, S_\mathsf{c}$
```

```
$\mathsf{validate}\big((B_0 \ldots B_n), \tau, T, t, s, N\big)$
**if** $\mathsf{len}\big((B_0 \ldots B_n)\big) = 0$ **then**
    **return**
        $\mathsf{timely}(\tau, T, t, \delta)$
        **return** $\tau - T \leq t + \delta$
**elseif** $B_0^i = N$ **then**
    **return** $\mathsf{timely}(\tau, B_n^\tau, t, \delta) \wedge$
        $\mathsf{seq}\big((B_0 \ldots B_n)\big)$
        $i \leftarrow B_0^i$
        **for** $B$ **in** $(B_1 \ldots B_n)$ **do**
            **if** $B^i \neq i + 1$ **then return** false
            $i \leftarrow i + 1$
        **return** true
**return** $\mathsf{timely}(\tau, B_n^\tau, t, \delta) \wedge \mathsf{seq}\big((B_0 \ldots B_n)\big) \wedge \mathsf{size}\big((B_0 \ldots B_n)\big) + B_0^{\ell_p} > s$
```