Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures*

Gunnar Hartung**, Björn Kaidel, Alexander Koch***, Jessica Koch***, and Dominik Hartmann***

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{gunnar.hartung, bjoern.kaidel, alexander.koch, jessica.koch}@kit.edu,
dominik.hartman@student.kit.edu

Abstract. Keeping correct and informative log files is crucial for system maintenance, security and forensics. Cryptographic logging schemes offer integrity checks that protect a log file even in the case where an attacker has broken into the system.

A relatively recent feature of these schemes is resistance against truncations, i.e. the deletion and/or replacement of the end of the log file. This is especially relevant as system intruders are typically interested in manipulating the later log entries that point towards their attack. However, there are not many schemes that are resistant against truncating the log file. Those that are have at least one of the following disadvantages: They are memory intensive (they store at least one signature per log entry), or fragile (i.e. a single error in the log renders the signature invalid and useless in determining where the error occurred).

We obtain a publicly-verifiable secure logging scheme that is simultaneously robust, space-efficient and truncation secure with provable security under simple assumptions. Our generic construction uses forward-secure signatures, in a plain and a sequential aggregate variant, where the latter is additionally fault-tolerant, as recently formalized by Hartung et al. (PKC 2016). Fault-tolerant schemes can cope with a number of manipulated log entries (bounded a priori) and offer strong robustness guarantees while still retaining space efficiency. Our implementation and the accompanying performance measurements confirm the practicality of our scheme.

Keywords: Sequential Aggregate Signatures · Fault-Tolerance · Secure Logging · Truncation-Security · Forward-Security

** The project underlying this report was supported by the German Federal Ministry of Education and Research under Grant No. 01|S15035A. The responsibility for the contents of this publication lies with the author.

*** This work was supported by the German Federal Ministry of Education and Research within the framework of the project KASTEL_IoE in the Competence Center for Applied Security Technology (KASTEL).

1 Introduction

Log files are an indispensable source of information for administrators investigating incidents in a computer system. They provide fine-grained information on actions and events that happened within the system, such as business transactions, errors, or security violations. Attackers frequently modify log files to cover their traces, so being able to distinguish real and faked information is crucial.

Therefore, the need to detect modifications to log files is widely recognized among computer security professionals, and much effort has been devoted to finding solutions that unveil such modifications (see below). Cryptographic solutions must be resilient to attackers that gain full control of the log server which holds the secret key. Thus, a secure logging scheme must stay secure *even if the attacker obtains the secret key* at some point in time, and must continue to enable the discovery of illicit log changes which occurred before the secret key was stolen by the attacker. This protects old log entries from unnoticed modification.

As this is impossible with standard authentication schemes, Anderson [1] (later formalized in [2], as remarked in [6]) proposed *forward-secure* schemes. These schemes assume that time is divided into intervals, called *epochs*, and use distinct secret keys for all epochs. For efficiency, we require that the secret key for an epoch t is computed from the secret key of the previous epoch $t-1$, and that there is a *single* verification key. By securely erasing secret keys when they expire, one ensures that an attacker cannot forge signatures for previous epochs.

Detecting *log truncations* is a surprisingly hard problem, because any authentication information computed by the log server can only authenticate past entries, and so there is nothing that authenticates the end of such a chain. Ma and Tsudik [22] were the first to present a mechanism to detect truncations of log files. Their solution is based on forward-secure *sequential aggregate signatures*. These signature schemes allow to "integrate" a signature for a new message into an already existing signature, which still has the size of a single signature, but authenticates all aggregated messages simultaneously.

The core property of their solution lies in the fact that for specific (sequential) aggregate signature schemes (such as [4]), removing a message from a given aggregate signature is intractable under standard assumptions. Ma and Tsudik [22] use this property by keeping only a single signature for the entire log file, which is an aggregate of the signatures for each individual message. The hardness of removing an individual signature then guarantees that no attacker can remove *any* message from the log file without notice, as in truncation attacks.

Problems with Existing Solutions. The approach by [22] is fragile, i.e. a single erroneous log entry renders the signature invalid, without any information on where the error is located. In a following investigation, distinguishing between real and faked information is no longer possible.

Only two solutions proposed so far are robust and truncation-secure at the same time, namely the second immutable scheme in [22], and the scheme of Hartung [8]. However, this robustness property is "bought" by falling back to

single signatures for each log entry, resulting in a very large log signature that is linear in the log size. Moreover, the truncation security of the former scheme is only argued informally, lacking a rigorous proof.

Our Solution. We propose a solution that overcomes these problems. The theoretical part of this paper formalizes a well-motivated security notion for secure logging and provides a provably-secure generic construction combining the fault-tolerance approach of [9] with the non-robust construction of [22], and add a single forward-secure signature on the log length for truncation security. This results in a scheme that is publicly-verifiable (as defined in [10]), and simultaneously features short signatures, robustness and truncation resistance.

We employ a recent technique from [9] to construct so-called *fault-tolerant* aggregate signatures, port this technique to the world of *sequential* aggregate signatures, and wed it with forward-security required for securing log files, which might be of independent interest. This is because *sequential* aggregate signatures are easier to obtain, often more efficient than ordinary aggregate signatures, and fully sufficient to realize secure log files with short signatures.

The technique from [9] also features so-called selective verification: To verify a single log entry, one can use the signature's redundancy to call the verification routine on a smaller set, instead of the whole log file, see [9, Sect. 4.2]. (Space-inefficient logs using single signatures have this feature trivially.)

Our approach is provably secure and uses a tight security reduction. For this, we define a security model for the logging scenario that captures truncation attacks as well as a wide range of other manipulations. This distinguishes our work from previous publications [23, 20] where truncation security is only argued for informally and it is not part of the security model.

We provide a secure logging scheme that can run on a stand-alone server without any interaction with another party. Our system does not require public ledgers (e.g. blockchains) or any other third party that needs to vouch for the integrity of the log file. However, our scheme can easily be combined with such techniques, and thus can be re-used as a building block for future schemes.

Contribution. In our work, we

- discuss why the notion of fault-tolerance from [9] is not applicable to the case of *sequential* aggregate signatures, give an alternative definition that also captures addition or removal of messages, adapt the generic construction of fault-tolerant aggregate signatures from [9] to the sequential aggregate case, prove its security, and prove its fault-tolerance w.r.t. our new definition,
- give a realistic and strong security notion for secure logging (similar to [8]) that also captures truncation attacks,
- give a generic construction of a publicly-verifiable robust secure logging scheme, which is space efficient and has a tight security reduction,
- present benchmark results based on a prototypical implementation of our scheme for multiple sets of parameters.

Related Work. Forward-secure signatures were first introduced by Bellare and Miner [2]. Many subsequent works followed, for example by Krawczyk [15] and Itkis and Reyzin [13]. Based on these works, Ma and Tsudik [20, 21] first considered forward security for sequential aggregate signatures.

Schneier and Kelsey [24] presented a logging scheme based on forward-secure MACs. Their scheme includes encryption of log entries to preserve confidentiality. Crosby and Wallach [7] presented a log scheme which allows for secure deletion of log entries without sacrificing the verifiability of the whole log. Their scheme relies on frequent interactions between a log server and several auditors. PillarBox [5] is a logging system focusing on additional properties such as confidentiality of log entries and logging rules. They assume forward-secure MACs as a tool, and use interaction to obtain truncation security. The only works that consider truncation security in a non-interactive setting are [10], [22] and [8], where only the latter takes a formal approach. The SALVE scheme in [8] additionally supports the secure generation of log excerpts which can be verified w.r.t. correctness and completeness for the excerpt criterion, without revealing log entries not in the excerpt. This scheme, [10] and the robust variants in [22] achieve robustness by recording one signature per message. In this paper, we obtain robustness without storing a signature for each log entry (saving storage space and potentially transmission bandwidth), while treating truncation security in a rigorous and formal manner.

2 Preliminaries

We define $[n] \coloneqq \{1, \ldots, n\}$. For vectors/tuples v, $v[i]$ denotes its i-th entry. If M is a matrix, $\mathrm{rows}(M)$ and $\mathrm{cols}(M)$ denote its number of rows and columns. $M[i, j]$ is the entry in the i-th row and j-th column of M.

The *security parameter* is denoted by $\kappa \in \mathbb{N}$. A probabilistic algorithm $\mathcal{A}$ is probabilistic polynomial time (PPT) if its running time is polynomial in κ. All algorithms are implicitly given 1^κ as input, even when not stated explicitly.

For $m_1, m_2 \in \{0,1\}^*$, $m_1 \,\|\, m_2$ denotes the *concatenation* of m_1 and m_2. For technical reasons, we assume that m_1 and m_2 can be uniquely derived from $m_1 \,\|\, m_2$. We use the same symbol for the concatenation of sequences, i.e. let $n, n' \in \mathbb{N}$ and $C = (c_1, \ldots, c_n)$ and $C' = (c'_1, \ldots, c'_{n'})$ be two sequences, then $C \,\|\, C' \coloneqq (c_1, \ldots, c_n, c'_1, \ldots, c'_{n'})$. If C' is a sequence with only one element c', we abbreviate this as $C \,\|\, c'$. If $C = (c_1, \ldots, c_n)$, then $|C| = n$.

2.1 Aggregate Signatures

Aggregate signature schemes were introduced by Boneh et al. [4]. Aggregate signatures can "combine" signatures from different signers on different messages into one single signature of equal size, authenticating all messages at once.

In their scheme, a signature is a single element of a group with a bilinear map. The aggregate of several signatures is their product in the group. Therefore,

aggregation is very flexible: signatures can be aggregated in any order, and aggregated signatures can be aggregated further.

Sequential aggregate signatures [19] do not support this fully flexible aggregation: Messages are added to an aggregate one-by-one, each message by its signer. Signing and aggregation may be a single, inseparable process (i.e., once created, signatures cannot in general be combined further). Sequential aggregate signatures are not as flexible, but are still useful in a wide range of applications, such as certificate chains, version control systems, and securing log files [22].

Claims and Claim Sequences. A *claim* $c = (\mathsf{pk}, i, m)$ is a triple of a public key pk, an epoch number $i \in \mathbb{N}_0$, and a message m. It conveys the meaning that the owner of pk has signed the message m during epoch i.[1]

A *claim sequence* is a finite sequence of claims. The *empty signature* λ is a signature valid for only the empty claim sequence $()$. Let $C = (c_1, \ldots, c_n)$ be a claim sequence and $b \in \{0,1\}^*$ with $|b| \geq n$ a bit sequence specifying a selection of indices. Then $C[b]$ is the subsequence of C containing the elements c_j $(1 \leq j \leq n)$ where $b[j] = 1$. If $\mathcal{M}$ is a matrix with only 1 and 0 entries, then $C[\mathcal{M}_i]$ is the subsequence containing all c_j, where $\mathcal{M}[i,j] = 1$, for $i \in [\mathrm{rows}(\mathcal{M})]$.

2.2 Forward-Secure Signatures

A forward-secure signature scheme [2] uses distinct secret keys for signing in each time interval (epoch). Throughout this paper we assume w.l.o.g. that the current epoch number can be efficiently derived from the current secret key.

Definition 1. *A* forward-secure signature *(FSS) scheme is a tuple of PPT algorithms* $\mathsf{FS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$*, where*

- $\mathsf{KeyGen}(1^\kappa, 1^T)$ *takes as input the security parameter* κ *and an a priori upper bound* T *on the number of epochs. It outputs a key pair* $(\mathsf{pk}, \mathsf{sk}_0)$*, where* sk_0 *is the secret key for the first epoch.*
- $\mathsf{Update}(\mathsf{sk}_t)$ *takes as input the secret key* sk_t *of period* t*. If* $t \geq T-1$ *its output is not defined. If* $t < T-1$ *it computes the secret key* sk_{t+1} *for the following period* $t+1$*. It then securely erases the old secret key* sk_t*.*
- $\mathsf{Sign}(\mathsf{sk}_t, m)$ *takes as input a secret key* sk_t *and a message* $m \in \{0,1\}^*$ *and outputs a signature* σ *for claim* (pk, t, m)*, where* t *is the epoch of* sk_t*.*
- $\mathsf{Verify}((\mathsf{pk}, t, m), \sigma)$ *outputs* 1 *if* σ *is a valid signature for the message* m *in epoch* t *under public key* pk*, and* 0 *otherwise.*

A FSS scheme is *correct* if any regularly signed message is valid, i.e. if for all epoch bounds $T = \mathrm{poly}(\kappa)$, all indices $t \in \{0, \ldots, T-1\}$ and all messages $m \in \{0,1\}^*$, it holds that $\mathsf{Verify}((\mathsf{pk}, t, m), \mathsf{Sign}(\mathsf{sk}_t, m)) = 1$, where $(\mathsf{pk}, \mathsf{sk}_0) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$ and $\mathsf{sk}_{t+1} = \mathsf{Update}(\mathsf{sk}_t)$ for $t = \{0, \ldots, T-2\}$.

[1] The terms "claim" and "claim sequence" are borrowed from [9]. However, we have added an epoch index i to each claim, because we are considering forward security in this work.

Security Notion for Forward-Secure Signatures. The security experiment for forward-secure signatures consists of four phases and is based on [2]. The general idea is that an adversary should not be able to forge a signature for any earlier epoch, even if he knows the secret key of the current epoch.

- *Setup Phase.* The challenger $\mathcal{C}$ generates a key pair $(\mathsf{pk}^*, \mathsf{sk}_0^*) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$ (where T is the maximal number of epochs) and gives the public key pk^* and T to the adversary. It sets $t := 0$.
- *Query Phase.* The adversary $\mathcal{A}$ has access to an Update and a Sign oracle. When $\mathcal{A}$ calls the Update oracle, $\mathcal{C}$ computes $\mathsf{sk}_{t+1}^* := \mathsf{Update}(\mathsf{sk}_t^*)$, sets $t := t + 1$, and returns "ok". $\mathcal{A}$ may only make $T - 1$ Update queries. $\mathcal{A}$ may (adaptively) issue signature queries to the Sign oracle for messages $m \in \{0,1\}^*$. For these queries, the challenger responds with a signature $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_t^*, m)$.
- *Break In Phase.* $\mathcal{A}$ may send a *break in* request to obtain the current secret key. $\mathcal{C}$ sets $t_{\mathsf{BreakIn}} := t$ and sends sk_t to $\mathcal{A}$. Afterwards, $\mathcal{A}$ is denied any further access to his oracles. We set $t_{\mathsf{BreakIn}} := \infty$ if $\mathcal{A}$ does not break in.
- *Forgery Phase.* Finally, $\mathcal{A}$ outputs a claim $(\mathsf{pk}^*, t^*, m^*)$ and a corresponding signature σ^*.

The adversary *wins* the experiment iff σ^* is a valid signature for claim $(\mathsf{pk}^*, t^*, m^*)$, and it is *non-trivial*, which means that m^* was not queried to the Sign oracle during period t^*, and $t^* < t_{\mathsf{BreakIn}}$.

A FSS scheme is *forward-secure existentially unforgeable under chosen message attacks* (FS-EUF-CMA-secure) if for each $T = \mathrm{poly}(\kappa)$ any PPT adversary $\mathcal{A}$ wins the above experiment with a probability that is at most negligible in κ.

2.3 Forward-Secure Sequential Aggregate Signatures

The following definition is the forward-secure sequential aggregate signature definition in [20], which is based on [2] and [19].

Definition 2. *A* forward-secure sequential aggregate signature *(FS-SAS) scheme is a tuple of four PPT algorithms* $\mathsf{AS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$*, where*

- $\mathsf{KeyGen}(1^\kappa, 1^T)$ *takes as input the security parameter* κ *and an a priori upper bound* T *on the number of epochs. It generates and outputs a key pair* $(\mathsf{pk}, \mathsf{sk}_0)$*, where* sk_0 *is the initial secret key for the first epoch.*
- $\mathsf{Update}(\mathsf{sk}_t)$ *takes as input the secret key* sk_t *of period* t*. If* $t \geq T - 1$ *the output of* Update *is not defined. If* $t < T - 1$ *it computes the secret key* sk_{t+1} *for the following period* $t + 1$*. It then securely erases the old secret key* sk_t*.*
- $\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i)$ *takes as input a secret key* sk_t *for an epoch* t*, a claim sequence* C_{i-1}*, a corresponding signature* σ_{i-1} *and a message* m_i*. It outputs a signature* σ_i *for the new claim sequence* $C_i := C_{i-1} \,\|\, (\mathsf{pk}, t, m_i)$*.*
- $\mathsf{Verify}(C, \sigma)$ *takes as input a claim sequence* C *and a signature* σ *and outputs* 1 *if* σ *is valid for* C*, and* 0 *otherwise.*

Informally, a signature is regular if it was generated with the correct use of the algorithms of a FS-SAS scheme. Formally, let C_i be a claim sequence and σ_i a signature. We say that σ_i is *regular* for C_i iff either $C_i = ()$ and $\sigma_i = \lambda$, or $C_i = C_{i-1} \| (\mathsf{pk}, t, m_i)$ and $\sigma_i \leftarrow \mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i)$ where σ_{i-1} is a regular signature for C_{i-1}, m_i is an arbitrary message, $(\mathsf{pk}, \mathsf{sk}_0)$ is a key pair output by $\mathsf{KeyGen}(1^\kappa, 1^T)$, and $\mathsf{sk}_{t+1} = \mathsf{Update}(\mathsf{sk}_t)$, for $t \in \{0, \ldots, T-1\}$.

A FS-SAS scheme is *correct* if for all bounds on the number of epochs $T = \text{poly}(\kappa)$, any signature σ which is regular for C is also valid for C.

Security Notion for FS-SAS Schemes. The security experiment for forward-secure sequential aggregate signatures in [20] consists of four phases and combines the experiments of forward-security [2] and sequential aggregate signatures [19].

- *Setup Phase.* The challenger generates a key pair $(\mathsf{pk}^*, \mathsf{sk}_0^*) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$, where T is the maximal number of time periods and gives the public key pk^* and T to the adversary. It sets $t := 0$.
- *Query Phase.* Here, the adversary $\mathcal{A}$ has access to an Update and an $\mathsf{AggSign}$ oracle. When $\mathcal{A}$ calls the Update oracle, the challenger computes $\mathsf{sk}_{t+1}^* := \mathsf{Update}(\mathsf{sk}_t^*)$, sets $t := t + 1$, and returns "ok". $\mathcal{A}$ is not allowed to make more than $T - 1$ queries to this oracle. The $\mathsf{AggSign}$ oracle takes as input a claim sequence C_{i-1}, a corresponding signature σ_{i-1} and a message m_i. It responds with $\sigma_i \leftarrow \mathsf{AggSign}(\mathsf{sk}_t^*, C_{i-1}, \sigma_{i-1}, m_i)$, where sk_t^* is the secret key for the current period t.
- *Break In Phase.* The adversary may send a *break in* request to obtain the current secret key. In this case, the experiment sets $t_{\mathsf{BreakIn}} := t$ and sends the current secret key sk_t for period t to $\mathcal{A}$. After $\mathcal{A}$ has broken in, he is denied any further access to his oracles. We set $t_{\mathsf{BreakIn}} := \infty$ if $\mathcal{A}$ does not break in.
- *Forgery Phase.* Finally, $\mathcal{A}$ outputs a claim sequence C^* and a corresponding signature σ^*.

The adversary *wins* the experiment iff σ^* is a valid signature for C^*, and C^* is *non-trivial*, i.e., C^* contains a claim $(\mathsf{pk}^*, t^*, m^*)$ for which $t^* < t_{\mathsf{BreakIn}}$ and $\mathcal{A}$ did not query m^* at its $\mathsf{AggSign}$ oracle during epoch t^*. A FS-SAS scheme is *forward-secure sequential aggregate signature existentially unforgeable under chosen message attacks* (FS-SAS-EUF-CMA-secure) if for each $T = T(\kappa) \in \text{poly}(\kappa)$ all PPT adversaries $\mathcal{A}$ win the above experiment only with a probability that is negligible in κ.[2]

2.4 Cover-Free Families

Cover-free families [14] are a combinatorial structure that allows us to achieve fault-tolerance in our constructions, as in [9]. Let $\mathcal{S}$ be a finite set, $\mathcal{B}$ be a set of

[2] This security notion is slightly weaker with respect to the non-triviality of forgeries than the one for sequential aggregate signatures by Lysyanskaya et al. [19]. There, they allow for all messages in C^* to be already queried before, but in different order. However, our notion additionally considers forward security.

subsets (or *blocks*) of $\mathcal{S}$ and $d \in \mathbb{N}$. The pair $\mathcal{F} = (\mathcal{S}, \mathcal{B})$ is a *d-cover-free family* if for any d blocks $B_1, \ldots, B_d \in \mathcal{B}$ and any distinct $B \in \mathcal{B} \setminus \{B_1, \ldots, B_d\}$, we have that $B \not\subseteq B_1 \cup \cdots \cup B_d$, i.e. no block is covered by the union of any other d blocks of $\mathcal{B}$. $\mathcal{F}$ is a *cover-free family (CFF)* if it is d-cover-free for a $d \geq 1$.

A CFF with a linear order $\leq$ on $\mathcal{B}$ is called *ordered*. To simplify the presentation, we also assume an order on $\mathcal{S}$ and usually identify $\mathcal{S}$ with $[r]$, for $r = |\mathcal{S}|$ in this case. The *incidence matrix* $\mathcal{M}$ of an ordered CFF is defined via

$$\mathcal{M}[i,j] = \begin{cases} 1, & \text{if } i \in B_j, \\ 0, & \text{otherwise,} \end{cases}$$

for $i \in [r] = \mathcal{S}$, $\mathcal{B} = \{B_1 \leq \cdots \leq B_m\}$ and $j \in [m]$.

We denote the i-th row of $\mathcal{M}$ by $\mathcal{M}_i \in \{0,1\}^m$. In this way, the rows of the matrix represent the elements of $\mathcal{S}$ and the columns represent the elements of $\mathcal{B}$.

3 Fault-Tolerant Forward-Secure Sequential Aggregate Signatures

In this section we define the syntax of forward-secure (multi-key) sequential aggregate signatures (SAS) with fault tolerance, discuss why the definition of fault-tolerance from [9] is not applicable in our case, and give an alternative definition. We then present a security notion that captures the forward-security property and is compatible with fault-tolerant sequential aggregate signatures, give a construction of such a scheme, and prove its fault-tolerance and its security.

Definition 3. *A key-evolving SAS scheme with list-verification Σ is a tuple of four PPT algorithms $\Sigma = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$, where:*

- $\mathsf{KeyGen}(1^\kappa, 1^T)$ *takes as input the security parameter κ and an upper bound T on the number of epochs. It outputs a key pair $(\mathsf{pk}, \mathsf{sk}_0)$, where sk_0 is the secret key for the first epoch.*
- $\mathsf{Update}(\mathsf{sk}_t)$ *takes as input the secret key sk_t of period t. If $t \geq T-1$ the output of* Update *is not defined. If $t < T-1$ it computes the secret key sk_{t+1} for period $t+1$ and securely erases the old key sk_t irrecoverably.*
- $\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i)$ *takes as input a secret key sk_t for an epoch t, a claim sequence C_{i-1}, a corresponding signature σ_{i-1} and a message m_i. It outputs a signature σ_i for the new claim sequence $C_i := C_{i-1} \,\|\, (\mathsf{pk}, t, m_i)$.*
- $\mathsf{Verify}(C, \sigma)$ *takes as input a claim sequence C of length $n \in \mathbb{N}_0$ and a signature σ for C and outputs a sequence V (also of length n) of claims and error symbols $\perp$. We require that for each $i \in [n]$, either $V[i] = C[i]$ or $V[i] = \perp$. (In other words, V can be obtained from C by replacing claims with $\perp$.) Claims output by* Verify *are taken to be valid.*

Let C_i be a claim sequence and τ_i be a signature. We say that τ_i is *regular* for C_i iff either $C_i = ()$ and $\tau_i = \lambda$, or $C_i = C_{i-1} \,\|\, (\mathsf{pk}, t, m_i)$ and $\tau_i \leftarrow \mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i)$ where τ_{i-1} is a regular signature for C_{i-1},

m_i is an arbitrary bit string, $t \in \{0, \ldots, T-1\}$, $T = T(\kappa) \in \text{poly}(\kappa)$, and sk_t is the t-times updated version of some secret key sk_0 such that $(\mathsf{pk}, \mathsf{sk}_0)$ is a key-pair output by $\mathsf{KeyGen}(1^\kappa, 1^T)$. We say that a SAS scheme with list verification is *correct*, if it is *0-fault-tolerant*, as defined in the next section.

3.1 Fault Tolerance of FS-SAS Schemes

Let $C = (c_1, \ldots, c_n), C' = (c'_1, \ldots, c'_{n'})$ be claim sequences. We say that C and C' *differ on ℓ positions* ($0 \leq \ell \leq \min(n, n')$) iff $c_i \neq c'_i$ for ℓ indices $1 \leq i \leq \min(n, n')$ and $c_i = c'_i$ for the rest. Moreover, we say that C' *contains d errors* with respect to C iff they differ on ℓ positions and $d = |n - n'| + \ell$.

A key-evolving SAS scheme Σ with list verification is *tolerant against d errors*, if for all claim sequences C, C', such that C' contains at most d errors with respect to C and for all signatures τ that are regular for C, we have

$$V[i] = c_i \quad \text{for all } 1 \leq i \leq \min(n, n') \text{ where } c_i = c'_i,$$

where $V \leftarrow \Sigma.\mathsf{Verify}(C', \tau)$. In other words, Verify outputs *at least* all claims c_i from C that C' did not modify. (It may also output claims where $C[i] \neq C'[i]$, but our security proof will show that such events are extremely rare or trivial.)

A *d-fault-tolerant key-evolving SAS scheme* is an SAS scheme with list verification that is tolerant against d errors. A scheme is *fault-tolerant*, if it is d-fault-tolerant for some $d > 0$.

On the Definition of Fault-Tolerance. In [9] a *multiset* of claim–signature pairs (c_i, τ_i) is said to contain d errors if d signatures τ_i are not regular for their respective claim c_i. This definition is not applicable to *sequential* aggregate signatures due to the lack of individual signatures τ_i. A natural approach that comes to mind is to define the number of errors via "intermediate" claim sequences $C_i = (c_1, \ldots, c_i)$ and their respective signatures τ_i. (This might not even be well-defined, but let us ignore this problem for the moment.) Following this approach, one might say that a claim sequence C contains d errors iff d of the signatures τ_i are not regular outputs of $\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i)$.

This approach fails, however, as it does not distinguish between signatures τ_i, which are partially damaged but sufficiently intact to authenticate some of the claims, and signatures that are completely destroyed. For example, consider the claim sequence $C = (c_1, \ldots, c_n)$ and the signatures $\tau_1, \ldots, \tau_n$, where all τ_i for $1 \leq i < n$ are regular for the respective intermediate claim sequence C_i, but τ_n is completely random. Then there was only one irregular step, and hence only one error with regard to this definition, but $\mathsf{Verify}(C, \tau_n)$ will output $(\perp, \ldots, \perp)$.

An alternative way to look at this is to observe that [9] implicitly assumes that the aggregation is correct, while errors only occur during signing. In the sequential aggregate case these two operations are inseparable in general, and we cannot assume that the aggregation did not introduce additional errors.

We therefore restrict our attention to specific changes to the claim sequence C: replacements of individual claims as well as addition or removal of claims at the

end of the sequence. These changes closely model our secure-logging scenario, as they capture events where an attacker edits log entries after breaking in, or removes tail-end log messages. (Note that addition or removal of claims is not considered in [9].)

3.2 Security Notion

Let $\mathsf{AS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$ be a key-evolving SAS scheme with list verification, $\mathcal{A}$ be a PPT algorithm, $\kappa \in \mathbb{N}$ a security parameter, and T be the number of epochs. The security experiment for a forward-secure SAS scheme with list verification is identical to that of forward-secure SAS schemes described in Sect. 2.3. The adversary $\mathcal{A}$ *wins* the experiment iff $\mathsf{Verify}(C^*, \tau^*)$ contains a claim c^* such that $c^* = (\mathsf{pk}^*, t^*, m^*)$ for some $m^* \in \{0,1\}^*$ and $t^* < t_{\mathsf{BreakIn}}$, and c^* is *non-trivial* in the sense that m^* was not given as an input to the $\mathsf{AggSign}$ oracle during epoch t^*.

A key-evolving SAS scheme with list verification is called *forward-secure sequential aggregate signature existentially unforgeable under chosen message attacks* (FS-SAS-EUF-CMA-secure) if for all $T = T(\kappa) \in \mathrm{poly}(\kappa)$, the probability of each PPT adversary $\mathcal{A}$ to win the above experiment is negligible in κ.

We say that a key-evolving SAS scheme with list verification is *single-key* FS-SAS-EUF-CMA-secure, if the above holds for all PPT adversaries $\mathcal{A}$ that never output claim sequences (to the signature oracle or as the forgery C^*) that contain a claim $c = (\mathsf{pk}, t, m)$ for a public key $\mathsf{pk} \neq \mathsf{pk}^*$. Clearly, a FS-SAS-EUF-CMA-secure scheme is also single-key FS-SAS-EUF-CMA-secure.

3.3 Generic Construction

We claim that the generic construction of [9] preserves the forward-security property of the underlying signature scheme. We use it to convert a forward-secure SAS scheme FSSAS to a fault-tolerant forward-secure SAS scheme.

Let FSSAS be a forward-secure SAS scheme, $\mathcal{F}$ a d-cover-free family ($d \in \mathbb{N}_0$), and $\mathcal{M}$ its incidence matrix. A signature in the new scheme is a vector of signatures of FSSAS. The algorithms of our scheme are as follows:

- KeyGen and Update are identical to the respective algorithms of FSSAS.
- $\mathsf{AggSign}(\mathsf{sk}_t, C_{j-1}, \tau_{j-1}, m_j)$ takes as input a secret key sk_t, a claim sequence $C_{j-1} = (c_1, \ldots, c_{j-1})$, its corresponding signature τ_{j-1} and a message m_j to sign. The sequential aggregate signature is updated component-wise, according to the entries of $\mathcal{M}$. More precisely, we set

$$\tau_j[i] \leftarrow \mathsf{FSSAS.AggSign}(\mathsf{sk}_t, C_{j-1}[\mathcal{M}_i], \tau_{j-1}[i], m_j),$$

  where $\mathcal{M}[i,j] = 1$, and let $\tau_j[i] \coloneqq \tau_{j-1}[i]$ otherwise ($i \in [\mathrm{rows}(\mathcal{M})]$). Here, $C_0 \coloneqq ()$ and $\tau_0[i] \coloneqq \lambda$ for each i. The output is τ_j.
- $\mathsf{Verify}(C, \tau)$ takes as input a claim sequence C of length $n \in \mathbb{N}_0$ and an aggregate signature τ for C. We compute a bit vector $b \in \{0,1\}^n$ that specifies

for each claim if it can safely be considered valid. For this, let $v|_\ell$ denote the vector v, truncated to the first ℓ elements. We initialize b to 0^n, and iterate over all entries $\tau[i]$ of τ, letting $b \leftarrow b \vee \mathcal{M}_i|_n$ if $\mathsf{FSSAS.Verify}(C[\mathcal{M}_i], \tau[i]) = 1$ in each iteration. (Here, $\vee$ denotes the bitwise logical OR of two bitstrings.) Finally, we build the output sequence V component-wise, by letting

$$V[j] = \begin{cases} C[j], & \text{if } b[j] = 1, \\ \perp, & \text{otherwise,} \end{cases} \qquad \text{for all } j \in [n].$$

Theorem 1. *Let Σ be the key-evolving SAS scheme with list verification defined above. If Σ is based on a d-CFF $\mathcal{F} = (\mathcal{S}, \mathcal{B})$, then it is tolerant against d errors.*

Theorem 2. *Let* FSSAS *be a key-evolving SAS scheme, $\mathcal{F}$ be a cover-free family with incidence matrix $\mathcal{M}$, and Σ be the scheme from Sect. 3.3. If there exists a PPT algorithm $\mathcal{A}$ that breaks the security of Σ with success probability $\varepsilon_{\mathcal{A}}$, then there exists an attacker $\mathcal{B}$ that breaks the FS-SAS-EUF-CMA-security of* FSSAS *with success probability $\varepsilon_{\mathcal{B}} \geq \varepsilon_{\mathcal{A}}$.*

Due to space constraints, we only give proof sketches here. The complete proofs can be found in the full version. For fault-tolerance, observe that each message m_j is redundantly aggregated into several of the signatures $\tau[i]$, namely those where $\mathcal{M}[i, j] = 1$. If errors occur on at most d positions, verification of a certain subset of all rows will fail. However, this subset cannot cover the rows for any correct message due to the cover-freeness of $\mathcal{F}$. Thus, each correct message can be verified from at least one row, and will therefore be output by our scheme.

For the security, note that our scheme essentially outputs the union of all messages that are contained in valid rows. Thus, to break the security, the attacker must create a signature where the target claim c^* is contained in at least one valid row, which constitutes a successful attack on the underlying scheme FSSAS.

4 Robust Secure Logging

In this section we introduce the notion of robust logging schemes and give a generic construction based on a plain forward-secure signature scheme, and a fault-tolerant forward-secure SAS scheme.

The syntax is as in FT-FS-SAS schemes, except that the key update algorithm may write to the log, and an additional error detection algorithm VerifyLog allows for fine-grained feedback on problems a log signature may have. This gives precise and reliable information on which parts of the log file are still trustworthy.

Definition 4. *A* logging scheme with list verification $\Lambda = (\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Update}, \mathsf{ValidEntries}, \mathsf{VerifyLog})$ *is a tuple of five PPT algorithms, where*

- $\mathsf{KeyGen}(1^\kappa, 1^T)$ *takes as input the security parameter κ and an a priori upper bound T on the number of epochs. It outputs a key pair $(\mathsf{pk}, \mathsf{sk}_0)$, where sk_0 is the secret key for the first epoch.*

– $\mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i)$ *takes as input a secret key* sk_t *for epoch* t*, a claim sequence* C_{i-1}*, a corresponding signature* σ_{i-1} *and a message* m_i*. It outputs a signature* σ_i *for the new claim sequence* $C_i := C_{i-1} \| (\mathsf{pk}, t, m_i)$*, thereby adding* m_i *to the log. (For efficiency, the public key is written just once into the log file in the single-key setting, instead of adding it to each log entry.)*
– $\mathsf{Update}(\mathsf{sk}_t, C, \sigma)$ *takes as input the secret key* sk_t *of period* t*. If* $t \geq T-1$ *the output is undefined. If* $t < T-1$ *it computes the secret key* sk_{t+1} *for period* $t+1$ *and securely erases the old key* sk_t*. The arguments* C *(a claim sequence) and* σ *(a signature) may be modified, e.g. to add epoch markers [3].*
– $\mathsf{ValidEntries}(C, \sigma)$ *takes as input a claim sequence* C *of length* $n \in \mathbb{N}_0$ *and a signature* σ *for* C *and outputs a sequence* V *(also of length* n*) of claims and error symbols* $\perp$*. We require that for each* $i \in [n]$*, either* $V[i] = C[i]$ *or* $V[i] = \perp$*. (I.e.,* V *can be obtained from* C *by replacing claims with* $\perp$*.) Claims output by* Verify *are taken to be valid.*
– $\mathsf{VerifyLog}(C, \sigma)$ *outputs either* $\emptyset$*, if the signature is without errors, or a subset of a set of error symbols* $\mathcal{E}$*, otherwise. We set* $\mathcal{E} := \{\perp_{\text{sig}}, \perp_{\text{len}}, \perp_{\text{em}}\}$*, with the interpretation that* $\perp_{\text{sig}} \in \mathsf{VerifyLog}(C, \sigma)$ *iff the signature is not valid, i.e.* $\mathsf{ValidEntries}(C, \tau) \neq C$*. Moreover, if* $\perp_{\text{len}} \in \mathsf{VerifyLog}(C, \sigma)$*, the signature may have been truncated. Finally,* $\perp_{\text{em}} \in \mathsf{VerifyLog}(C, \sigma)$ *if some problem with epoch markers has been detected.*

Fault-tolerance is defined analogously to Sect. 3.1, substituting Append for $\mathsf{AggSign}$, and $\mathsf{ValidEntries}$ for Verify. A logging scheme with list verification is *robust* if it is fault-tolerant and we have that regular log files are error-free (i.e. $\mathsf{VerifyLog}(C, \sigma) = \emptyset$) and error-free log files are valid (i.e. $\mathsf{ValidEntries}(C, \tau) = C$). Note that, if the signature is valid in the sense that all claims are returned by $\mathsf{ValidEntries}$, it is still possible that an attacker might have truncated the log. In this case an error symbol returned by $\mathsf{VerifyLog}$ points towards this possibility.

The security notion for logging schemes is similar to the FS-EUF-CMA notion for FT-FS-SAS schemes, but models the real world setting of secure logging more closely: The log server maintains a state which the adversary influences only through his oracles. In more detail, a log append oracle appends an entry to the internal log file, and an adversary can never again add messages to any earlier state of the log file. Moreover, the internal signatures remain hidden from him by default, as these usually stay on the server.

To strengthen the notion, we introduce an additional oracle returning the current signature, which models a public verification of the log file by a third party. To exclude trivial attacks, we explicitly disallow an adversary to truncate the log file to a state he has gotten a signature for. However, he may try to use these signatures to, e.g., truncate the log file to a different previous state.

At the end of the experiment, the attacker outputs a forgery. We require error-freeness ($\mathsf{VerifyLog}(C^*, \sigma^*) = \emptyset$), as otherwise the adversary might use a combination of introducing faults and truncating the claim sequence to obtain a valid signature (verification of the forged signature and claim sequence outputs the full forged claim sequence) that violates other anti-truncation mechanisms.

Definition 5. *For a log scheme with list verification* $\Lambda = (\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Update}, \mathsf{ValidEntries}, \mathsf{VerifyLog})$*, a PPT adversary* $\mathcal{A}$*, the number of epochs* T *and the security parameter* $\kappa \in \mathbb{N}_0$*, the security experiment FS-EUF-CLMA*[3]*-*$Exp_{\Lambda,\mathcal{A},T}(\kappa)$ *is defined as follows:*

Setup Phase. *The experiment generates a key pair* $(\mathsf{pk}, \mathsf{sk}_0) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$*, the log file* $C_0 := ()$ *and signature* $\sigma_0 := \lambda$*. It initializes the epoch counter* $t := 0$*, and starts* $\mathcal{A}$ *with inputs* pk*,* T*.*

Query Phase. $\mathcal{A}$ *may adaptively issue queries to the following oracles:*

- **LogAppend Oracle.** *The experiment appends the specified message* m *to the log and updates the signature via* $\sigma_i \leftarrow \mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m)$*, where* σ_{i-1} *denotes the previous signature, and returns "ok".*
- **NextEpoch Oracle.** *The oracle updates the secret key, the log and its signature via* $\mathsf{Update}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1})$*, increments the epoch counter* $t := t + 1$ *and returns "ok". It may be queried at most* $T - 1$ *times.*
- **GetSignature Oracle.** *Whenever* $\mathcal{A}$ *calls the* GetSignature *oracle, the challenger responds with the current signature* σ_i *of the log.*

Break In Phase. *The adversary may break in to obtain the current secret key* sk_t*. If* $\mathcal{A}$ *does, the experiment sets* $t_{\mathsf{BreakIn}} := t$*. Otherwise, let* $t_{\mathsf{BreakIn}} := \infty$*.*

Forgery Phase. $\mathcal{A}$ *outputs a log file* C^**, and a forged signature* σ^* *for* C^**.*

We say that $\mathcal{A}$ wins *the experiment, iff the following conditions hold.*

- *The signature* σ^* *is* error-free*, i.e.* $\mathsf{VerifyLog}(C^*, \sigma^*) = \emptyset$*. (This implies that the signature is valid.)*
- *The signature is* non-trivial *as defined next. Let* C' *be the subsequence of* C^* *that is obtained by deleting all claims* $c = (\mathsf{pk}, t, m)$ *from* C^**, where* $t \geq t_{\mathsf{BreakIn}}$*.* $\mathcal{A}$*'s forgery is* non-trivial*, iff* $|C'| \neq 0$ *and* C' *does not equal the content of the log file during any* GetSignature *query.*

A logging scheme with list verification Λ *is said to be* FS-EUF-CLMA-secure*, iff for all* $T = T(\kappa) \in \mathrm{poly}(\kappa)$ *and all probabilistic polynomial time attackers* $\mathcal{A}$ *the probability for* $\mathcal{A}$ *winning the above experiment is negligible in* κ*.*

4.1 Generic Construction

We give a generic construction of a simultaneously secure and robust log scheme $\Lambda = (\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Update}, \mathsf{ValidEntries}, \mathsf{VerifyLog})$. Let AS be a key-evolving SAS scheme with list verification and FS a key-evolving signature scheme.

- $\mathsf{KeyGen}(1^\kappa, 1^T)$ creates key pairs of the underlying schemes AS and FS as $(\mathsf{pk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{AS}}) \leftarrow \mathsf{AS.KeyGen}(1^\kappa, 1^T)$, $(\mathsf{pk}_{\mathsf{FS}}, \mathsf{sk}_{\mathsf{FS}}) \leftarrow \mathsf{FS.KeyGen}(1^\kappa, 1^T)$ and returns $\mathsf{pk} = (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$ and $\mathsf{sk}_0 = (\mathsf{sk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{FS}})$.

[3] *forward-secure existentially unforgeable under chosen log message attacks*

– $\mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i)$ takes as input a secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{FS}})$ for period t, a claim sequence $C_{i-1} = (c_1, \ldots, c_{i-1})$, its corresponding signature $\tau_{i-1} = (\sigma_{i-1}, s_{i-1})$ and a message m_i to sign. Both signature components are obtained from the signature algorithms of AS and FS via

$$\sigma_i \leftarrow \mathsf{AS.AggSign}(\mathsf{sk}_{\mathsf{AS}}, C_{i-1}, \sigma_{i-1}, m_i \,\|\, i), \text{ and}$$
$$s_i \leftarrow \mathsf{FS.Sign}(\mathsf{sk}_{\mathsf{FS}}, i).$$

  Append securely erases the old length signature s_{i-1} so that it cannot be used in case of a later break in. The resulting signature $\tau_i = (\sigma_i, s_i)$ is returned.
– $\mathsf{Update}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1})$ takes as input the secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{FS}})$, a claim sequence C_{i-1} and a corresponding signature τ_{i-1}, and appends an epoch marker to the log file that is valid for the current epoch t, via

$$\tau_i \leftarrow \mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i),$$

  where $m_i = \texttt{"End of epoch:"} \,\|\, t$. It then updates the components of sk_t via $\mathsf{sk}'_{\mathsf{AS}} \leftarrow \mathsf{AS.\,Update}(\mathsf{sk}_{\mathsf{AS}})$ and $\mathsf{sk}'_{\mathsf{FS}} \leftarrow \mathsf{FS.\,Update}(\mathsf{sk}_{\mathsf{FS}})$. (These algorithms erase the old keys securely.) The new secret key is $\mathsf{sk}_{t+1} = (\mathsf{sk}'_{\mathsf{AS}}, \mathsf{sk}'_{\mathsf{FS}})$, the new claim sequence is $C_i = C_{i-1} \,\|\, (\mathsf{pk}, t, m_i)$, and the new signature is τ_i.
– $\mathsf{ValidEntries}(C, \tau)$ takes as input a claim sequence C and a signature $\tau = (\sigma, s)$ for C. It outputs $\mathsf{AS.Verify}(C', \sigma)$, where C' is the claim sequence generated from C by appending the message number i to m_i for all claims in C.
– $\mathsf{VerifyLog}(C, \tau)$ takes as input a claim sequence C and a signature $\tau = (\sigma, s)$ for C. It maintains an error set E initialized to $\emptyset$. Firstly, it verifies the FS signature s using $b = \mathsf{FS.Verify}((\mathsf{pk}_{\mathsf{FS}}, t, |C|), s)$. If $b = 0$, it adds $\perp_{\text{len}}$ to E. Then it proceeds with checking the epoch markers: For all claims $c_i = (\mathsf{pk}, t_i, m_i)$ and $c_{i+1} = (\mathsf{pk}, t_{i+1}, m_{i+1})$ in C, where $t_{i+1} \neq t_i$, consider two cases. If $t_{i+1} \neq t_i + 1$ then output $\perp_{\text{em}}$, else check if $m_i = \texttt{"End of epoch:"} \,\|\, t_i$, otherwise output $\perp_{\text{em}}$. Finally, it checks whether the signature is valid, i.e. $\mathsf{ValidEntries}(C, \tau) = C$, and adds $\perp_{\text{sig}}$ to E, if this is not the case. It outputs the set of errors E.

The log scheme Λ described above is d-fault-tolerant, if the underlying FT-FS-SAS scheme AS is d-fault-tolerant. We omit the proof due to space restrictions.

Theorem 3. *Our log scheme with list verification Λ is FS-EUF-CLMA-secure, if AS is FS-SAS-EUF-CMA-secure and FS is FS-EUF-CMA-secure. More precisely, for any PPT adversary $\mathcal{A}$ who breaks the FS-EUF-CLMA-security with success probability $\varepsilon_{\mathcal{A}}$, there exists a PPT adversary $\mathcal{B}$ who either breaks the FS-SAS-EUF-CMA-security of AS or the FS-EUF-CMA-security of FS with success probability at least $\varepsilon_{\mathcal{B}}^{\mathsf{AS}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$ and $\varepsilon_{\mathcal{B}}^{\mathsf{FS}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$, respectively.*

Let us first give some overview and intuition about the proof. To win the security experiment, an attacker $\mathcal{A}$ must either truncate the log file to a state he has not seen the signature for, or create a valid signature for a log file modified w.r.t. an epoch before his break-in. If $\mathcal{A}$ truncates the log file without detection, he must

create a new signature s for the length of the log file, which violates the security of FS. If $\mathcal{A}$ forges a signature for log file modified w.r.t. a previous epoch, then $\mathcal{A}$ has broken the security of AS. Since we assume that both base schemes are secure, our resulting construction must be secure, too.

Proof. A FS-EUF-CLMA-adversary $\mathcal{A}$ can adaptively query the three oracles *LogAppend*, *GetSignature* and *NextEpoch* before he may break in, and then outputs a forgery (C^*, τ^*), where $\tau^* = (\sigma^*, s^*)$. As any signatures appended after the break in are trivial to produce, let C'^* be the claim sequence after deleting all claims (pk, t, m) of C^*, where $t \geq t_{\mathsf{BreakIn}}$. Let $C^{\exp}$ be the internal claim sequence of the experiment, after $\mathcal{A}$ did his last *GetSignature* query in a period $t < t_{\mathsf{BreakIn}}$. We consider two different events:

- E_1 occurs, if (C^*, τ^*) is error-free, non-trivial and C'^* is not a prefix of $C^{\exp}$.
- E_2 occurs, if (C^*, τ^*) is error-free, non-trivial and C'^* is a prefix of $C^{\exp}$.

We have $\varepsilon_{\mathcal{A}} \leq \Pr[E_1] + \Pr[E_2]$ and thus $\Pr[E_1] \geq \frac{\varepsilon_{\mathcal{A}}}{2}$ or $\Pr[E_2] \geq \frac{\varepsilon_{\mathcal{A}}}{2}$. Please note that in the following paragraphs $\mathsf{sk}^i_{\mathsf{AS}}, \mathsf{sk}^i_{\mathsf{FS}}$ denote the secret keys of period i for the respective schemes.

Attack on the FS-SAS-EUF-CMA-security of AS. First we construct a FS-SAS-EUF-CMA-adversary $\mathcal{B}$ on AS, who uses a successful FS-EUF-CLMA-adversary $\mathcal{A}$ and has to simulate the FS-EUF-CLMA-security experiment for $\mathcal{A}$. The challenger in the FS-SAS-EUF-CMA-security experiment generates a key pair $(\mathsf{pk}_{\mathsf{AS}}, \mathsf{sk}^0_{\mathsf{AS}}) \leftarrow \mathsf{AS.KeyGen}(1^\kappa, 1^T)$ and sends $\mathsf{pk}_{\mathsf{AS}}$ and the maximal number of epochs T to $\mathcal{B}$. $\mathcal{B}$ uses FS to generate a key pair $(\mathsf{pk}_{\mathsf{FS}}, \mathsf{sk}^0_{\mathsf{FS}}) \leftarrow \mathsf{FS.KeyGen}(1^\kappa, 1^T)$, s.t. $\mathsf{pk} := (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$ and $\mathsf{sk}_0 := (\mathsf{sk}^0_{\mathsf{AS}}, \mathsf{sk}^0_{\mathsf{FS}})$ for the current period 0. $\mathcal{B}$ forwards pk and T to $\mathcal{A}$. $\mathcal{B}$ initializes the log and signature it maintains towards $\mathcal{A}$ as $C_0 := ()$, $\sigma_0 := \lambda$ and sets $i := 0$, $t := 0$ and $\mathcal{L}_{\mathrm{FS}} := \{\mathsf{sk}^0_{\mathsf{FS}}\}$.

We describe how $\mathcal{B}$ simulates the three oracles and the break in phase for $\mathcal{A}$:

LogAppend Oracle. $\mathcal{A}$ sends $\mathcal{B}$ a query m_i. $\mathcal{B}$ sets $C_i := C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, m_i \,\|\, i)$ for the current period t. $\mathcal{B}$ sends an $\mathsf{AggSign}$ query $m_i \,\|\, i$ with claim sequence C_{i-1} and signature σ_{i-1} to his challenger who responds with a signature σ_i. Finally, it sets $i := i + 1$ and sends $\mathcal{A}$ the string "ok".

NextEpoch Oracle. When $\mathcal{A}$ sends a *NextEpoch* query, $\mathcal{B}$ stops if $t \geq T - 1$ and outputs $\perp$, otherwise $\mathcal{B}$ sets $m_i :=$ `"End of epoch:"` $\| t$ and $C_i := C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, m_i \,\|\, i)$. $\mathcal{B}$ obtains the signature σ_i for C_i from his $\mathsf{AS.AggSign}$ oracle the same way as before. $\mathcal{B}$ sends an *Update* query to the challenger, who computes $\mathsf{sk}^{t+1}_{\mathsf{AS}} := \mathsf{AS.Update}(\mathsf{sk}^t_{\mathsf{AS}})$. $\mathcal{B}$ computes $\mathsf{sk}^{t+1}_{\mathsf{FS}} := \mathsf{FS.Update}(\mathsf{sk}^t_{\mathsf{FS}})$ by its own. $\mathcal{B}$ sets $i := i + 1$, $t := t + 1$ and $\mathcal{L}_{\mathrm{FS}} := \mathcal{L}_{\mathrm{FS}} \cup \{\mathsf{sk}^{t+1}_{\mathsf{FS}}\}$ and returns "ok".

GetSignature Oracle. When $\mathcal{A}$ calls the *GetSignature* oracle, $\mathcal{B}$ determines the length i of the current claim sequence C_i and the period t_{last} of the last claim in C_i, which is either the current period t or $t-1$ (since an epoch switch always adds an end-of-epoch claim). $\mathcal{B}$ gets $\mathsf{sk}^{t_{\mathrm{last}}}_{\mathsf{FS}}$ from $\mathcal{L}_{\mathrm{FS}}$ and computes $s_i \leftarrow \mathsf{FS.Sign}(\mathsf{sk}^{t_{\mathrm{last}}}_{\mathsf{FS}}, i)$. The new signature for C_i is now $\tau_i = (\sigma_i, s_i)$ and $\mathcal{B}$ sends τ_i to $\mathcal{A}$.

Break In Phase. When $\mathcal{A}$ breaks in, $\mathcal{B}$ sets $t_{\mathsf{BreakIn}} := t$ and sends his challenger also a *break in* request. $\mathcal{B}$ gets the current secret key $\mathsf{sk}_{\mathsf{AS}}^t$ and sends $\mathcal{A}$ the current secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}^t, \mathsf{sk}_{\mathsf{FS}}^t)$.

If event E_1 occurs, then $\mathcal{A}$ sends $\mathcal{B}$ an *error-free* and *non-trivial* signature $\tau^* = (\sigma^*, s^*)$ for a claim sequence C^*, where C'^* is not a prefix of $C^{\exp}$. Since τ^* is non-trivial, $i' := |C'^*| \neq 0$. In this case, there exists an index $j' \in [i']$, s.t. the claim $c_{j'}^* = (\mathsf{pk}, t_{j'}^*, m_{j'}^*) \neq c_{j'}^{\exp}$ and $t_{j'} < t_{\mathsf{BreakIn}}$. Let C' be the claim sequence that is generated by appending the message index i to m_i in each of the claims from C^*. Since $\mathcal{A}$'s forgery is valid, (C', σ^*) is also a valid forgery for $\mathcal{B}$'s challenger and it is non-trivial, because the claim $(\mathsf{pk}, t_{j'}^*, m_{j'}^* \,\|\, j')$ is fresh[4]. So $\mathcal{B}$ can forward this and therefore has success probability $\varepsilon_{\mathcal{B}}^{\mathsf{AS}} \geq \Pr[E_1]$.

Attack on the FS-EUF-CMA-security of FS. Next, we construct a FS-EUF-CMA-adversary $\mathcal{B}$ on FS, who uses a successful FS-EUF-CLMA-adversary $\mathcal{A}$ and has to simulate the FS-EUF-CLMA-security experiment for $\mathcal{A}$. The challenger in the FS-EUF-CMA-security experiment generates a key pair $(\mathsf{pk}_{\mathsf{FS}}, \mathsf{sk}_{\mathsf{FS}}^0) \leftarrow \mathsf{FS.KeyGen}(1^\kappa, 1^T)$ and sends $\mathsf{pk}_{\mathsf{FS}}$ and the maximal number of epochs T to $\mathcal{B}$. $\mathcal{B}$ uses the AS-scheme and generates a key pair $(\mathsf{pk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{AS}}^0) \leftarrow \mathsf{AS.KeyGen}(1^\kappa, 1^T)$ and forwards $\mathsf{pk} = (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$ and T to $\mathcal{A}$. $\mathcal{B}$ initializes the log and signature it maintains towards $\mathcal{A}$ as $C_0 := ()$, $\sigma_0 := \lambda$ and sets $i := 0, t := 0, t' := 0$. We describe how $\mathcal{B}$ simulates the three oracles and the break in phase for $\mathcal{A}$:

LogAppend Oracle. $\mathcal{A}$ sends $\mathcal{B}$ a query m_i. $\mathcal{B}$ sets $C_i := C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, m_i \,\|\, i)$ for the current period t and $\sigma_i \leftarrow \mathsf{AS.AggSign}(\mathsf{sk}_{\mathsf{AS}}, C_{i-1}, \sigma_{i-1}, m_i \,\|\, i)$. Then $\mathcal{B}$ sets $i := i + 1$ and sends $\mathcal{A}$ the string "ok".

NextEpoch Oracle. When $\mathcal{A}$ sends a *NextEpoch* query, $\mathcal{B}$ stops if $t \geq T - 1$ and outputs $\perp$, otherwise $\mathcal{B}$ sets $m_i := \texttt{"End of epoch:"} \,\|\, t$ and $C_i := C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, m_i)$, and computes σ_i in the same way as before. $\mathcal{B}$ computes $\mathsf{sk}_{\mathsf{AS}}^{t+1} := \mathsf{AS.Update}(\mathsf{sk}_{\mathsf{AS}}^t)$ by its own, sets $i := i + 1, t := t + 1$ and returns "ok".

GetSignature Oracle. When $\mathcal{A}$ calls the *GetSignature* oracle $\mathcal{B}$ determines the length i of the current claim sequence C_i and the period t_{last} of the last claim in C_i (t_{last} is either t or $t - 1$). If $t_{\text{last}} - t' := d \neq 0$, then $\mathcal{B}$ sends d *Update* queries to his challenger, who computes $\mathsf{sk}_{\mathsf{FS}}^{t_{\text{last}}}$ via updating the current $\mathsf{sk}_{\mathsf{FS}}^{t'}$ d times. Then $\mathcal{B}$ sends a query $m = i$ to his challenger, who responds with $s_i \leftarrow \mathsf{FS.Sign}(\mathsf{sk}_{\mathsf{FS}}^{t_{\text{last}}}, i)$. The new signature for C_i is now $\tau_i = (\sigma_i, s_i)$ and $\mathcal{B}$ sends τ_i to $\mathcal{A}$. $\mathcal{B}$ sets $t' := t$ and if $t - t_{\text{last}} = 1$, sends one more *Update* query.

Break In Phase. When $\mathcal{A}$ sends his *break in* request, $\mathcal{B}$ sets $t_{\mathsf{BreakIn}} := t$. If $t - t' := d \neq 0$, then $\mathcal{B}$ sends d *Update* queries to his challenger and then sends also a *break in* request. $\mathcal{B}$ gets the current secret key $\mathsf{sk}_{\mathsf{FS}}^t$ and sends $\mathcal{A}$ the current secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}^t, \mathsf{sk}_{\mathsf{FS}}^t)$.

[4] Remember that we assume that m and i can be uniquely derived from $m \,\|\, i$, which implies that the claims $c_{j'}^*$ and $c_{j'}^{\exp}$ also differ after concatenating j' to their messages. Since j' is also only used once, the claim $c_{j'}^*$ cannot become equal to any other claim of $C^{\exp}$ after this concatenation, either.

If event E_2 occurs, then $\mathcal{A}$ sends $\mathcal{B}$ an *error-free* and *non-trivial* signature $\tau^* = (\sigma^*, s^*)$ for a claim sequence C^*, where C'^* is a prefix of C^{exp}. Let $|C^*| =: i^*$, $|C'^*| =: i'$, $|C^{\text{exp}}| =: i^{\text{exp}}$. Thus, s^* is a valid signature for i^*. We show that $i^* < i^{\text{exp}}$ and i^* was not queried to $\mathcal{B}'$s challenger during the experiment before.

If $i^* \geq i^{\text{exp}}$, then $i' > i^{\text{exp}}$ is not possible, since C'^* is a prefix of C^{exp}. $i' < i^{\text{exp}}$ is possible neither, since the claim $c^*_{i'}$ must then be $(\mathsf{pk}, t_{i'}, m_{i'} = \texttt{"End of epoch:"} \,\|\, t_{i'})$ (since all claims where $t \geq t_{\mathsf{BreakIn}}$ were deleted, the last claim of C'^* must be a claim for an epoch marker – this follows from the error-freeness of τ^*) for $t_{i'} = t_{\mathsf{BreakIn}} - 1$. Because C'^* is a prefix of C^{exp} this is also a claim in C^{exp}. Since C^{exp} also contains no claims for any $t \geq t_{\mathsf{BreakIn}}$, this also has to be the last claim in C^{exp}. Therefore $|C'^*|$ must be equal to $|C^{\text{exp}}|$. Thus, $i' = i^{\text{exp}}$ and $C'^* = C^{\text{exp}}$, but in this case, τ^* is not a non-trivial signature, because $\mathcal{A}$ queried his *GetSignature* oracle for $C'^* = C^{\text{exp}}$ per definition.

So, we have $i^* < i^{\text{exp}}$ and therefore $C^* = C'^*$ since C^* contains no claims with $t \geq t_{\mathsf{BreakIn}}$ in this case. So, C^* is also a prefix of C^{exp}. Because τ^* is an error-free and non-trivial signature, $\mathcal{A}$ has never queried the *GetSignature* oracle when the internal state of the log was equal to C^*. Thus, $\mathcal{B}$ has never queried his $\mathsf{FS.Sign}$ oracle for i^*, so s^* is a fresh and valid signature of i^* under $\mathsf{sk}_{t_{i^*}}$, with $t_{i^*} < t_{\mathsf{BreakIn}}$. Thus, $\mathcal{B}$ forwards a valid forgery (i^*, s^*) to his challenger with success probability $\varepsilon^{\mathsf{FS}}_{\mathcal{B}} \geq \Pr[E_2]$. In total, we have $\varepsilon^{\mathsf{AS}}_{\mathcal{B}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$ or $\varepsilon^{\mathsf{FS}}_{\mathcal{B}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$. □

5 Implementation and Performance Results

We implemented our generic construction from Sect. 4.1 and conducted various benchmarks. Our scheme uses the BGLS-FS-SAS scheme [23, 20] and the BM-FSS scheme [2]. Our results are shown in Table 1. Details of our implementation and benchmarks are provided in Appendix A.

Methodology. For our experiments, we defined several sets of processes. Each process was repeated three times. The averages and standard deviations shown in Table 1 have therefore been computed from a sample of size 3.

For the first set of processes, we called the KeyGen algorithm with the given parameter T and measured its total run-time. In the second set, we created a random key for T epochs, and then measured the run-time of updating the key T times, without computing any signatures. Table 1 shows the average run-time per invocation of Update.

The third process consisted of creating a key-pair valid for n epochs, and then calling the $\mathsf{AggSign}$ algorithm n times, switching epochs every ℓ messages. For each epoch switch, we created and signed an epoch marker first, and then updated the secret key. The process also included signing the current counter value with a forward-secure digital signature scheme and updating that scheme. The time shown in Table 1 is the total time of all signing and updating operations, divided by the number of messages, so it represents the average time needed for adding a single log entry to the log file. The standard deviation was computed over the average signing time in each run.

Table 1. Runtimes of our robust secure logging schemes based on the BGLS-FS-SAS from [20]. See the methodology section for an explanation of this table and the full version for more data.

Algorithm	**Parameter**	ℓ	**avg [ms]**	**STD [ms]**
KeyGen	$T = 10\,000$		38 053	241
Update	$T = 10\,000$		18.6	0.033
AggSign + Update	$n = 1000$	10	67.5	0.014
	$n = 1000$	100	60.4	0.048
	$n = 1000$	1000	59.7	0.019
Verify	$n = 1000$	10	271	2.05
	$n = 1000$	100	227	1.87
	$n = 1000$	1000	22.5	0.15

The measurements in the last set of processes were obtained by calling Verify after a completion of a process from the third set. The time given in Table 1 is an average of the run-time of three executions divided by the number of messages that were verified. Hence, it represents the average verification time per message. The standard deviation was computed over the run-times of an individual execution divided by n. We did not consider invalid signatures in our experiments.

6 Conclusion

We give a simple solution to the problem of space-efficient logging, while still retaining robustness *and* truncation security for a properly formalized security notion of secure logging and achieve provable security. Combining a fault-tolerant forward-secure sequential aggregate signature with a forward-secure signature on the current log length elegantly solved these problems in combination. For this we modified the notion of fault-tolerance from [9] (which is based on cover-free families to introduce redundancy), to fit the more restricted setting of sequential aggregate signatures, allowing for more efficient implementations due to less requirements than in the case of general aggregation. Finally, we evaluated the performance of a prototype implementation of our space-efficient and truncation-resistant robust secure logging scheme.

A Implementation Details

This section gives details about our implementation of the scheme from Sect. 4.1. Our implementation is written in C++11, and will be made available under a free software license. For the BM-FSS scheme, we chose a modulus size of 1024 bits, roughly equivalent to a security level of 80 bit. The BGLS scheme was instantiated using elliptic curve groups 160 bits, and the base field had 1024 bits.

We used an instantiation of the cover-free family based on polynomials, described in [16]. For a CFF supporting $n = 100$, 1000, and 10000 messages, we chose the field size $q = 5$, 11, and 23, respectively, and fixed the polynomial degree at $k = 2$. This led to $d = 2, 5$ and 11, respectively. (The resulting CFFs were slightly larger than required: They supported 125, 1331, and 12167 messages, respectively.) Whenever a hash function was needed, we used SHA-256. We used a constant string of 200 bytes for all messages.

Our experiments were conducted on a laptop computer with an Intel Core i5-2430M CPU [12] with a clock rate of 2.4 GHz. (Our implementation is not parallelized and therefore did not make use of the additional processor cores.) The processor has private (per-core) caches of 128 KB (Level 1) and 512 KB (Level 2), and a shared Level 3 Cache of 3072 KB [11, Section 1.1] The system was equipped with 5.7 GiB of RAM and running a 64-bit version desktop version of the Fedora 23 GNU/Linux operating system, equipped with Linux Kernel version 4.4.9-300. All code was compiled with the GNU C Compiler (version 5.3.1) and optimization level set to `-O2`. We used Shoups NTL library [25] (version 9.4.0) for the implementation of the BM-FSS scheme and the PBC library [18] (version 0.5.14) for the implementation of the BGLS-FS-SAS scheme.

References

[1] R. Anderson. *Invited Lecture. 4th ACM Computer and Communications Security*. 1997.

[2] M. Bellare and S. K. Miner. "A Forward-Secure Digital Signature Scheme". In: *CRYPTO 1999*. Ed. by M. J. Wiener. Vol. 1666. LNCS. Springer, 1999, pp. 431–448. DOI: 10.1007/3-540-48405-1_28.

[3] M. Bellare and B. Yee. *Forward Integrity for Secure Audit Logs*. Tech. rep. Computer Science and Engineering Department, University of California at San Diego, 1997.

[4] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *EUROCRYPT 2003*. Ed. by E. Biham. Vol. 2656. LNCS. Springer, 2003, pp. 416–432. DOI: 10.1007/3-540-39200-9_26.

[5] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. "PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging". In: *Research in Attacks, Intrusions and Defenses, RAID 2014*. Ed. by A. Stavrou, H. Bos, and G. Portokalidis. Vol. 8688. LNCS. Springer, 2014, pp. 46–67. DOI: 10.1007/978-3-319-11379-1_3.

[6] X. Boyen, H. Shacham, E. Shen, and B. Waters. "Forward-secure signatures with untrusted update". In: *CCS 2006*. Ed. by A. Juels, R. N. Wright, and S. D. C. di Vimercati. ACM, 2006, pp. 191–200. DOI: 10.1145/1180405.1180430.

[7] S. A. Crosby and D. S. Wallach. "Efficient Data Structures For Tamper-Evident Logging". In: *USENIX 2009*. Ed. by F. Monrose. USENIX Association, 2009, pp. 317–334. URL: http://www.usenix.org/events/sec09/tech/full_papers/crosby.pdf.

[8] G. Hartung. "Secure Audit Logs with Verifiable Excerpts". In: *CT-RSA 2016*. Ed. by K. Sako. Vol. 9610. LNCS. Springer, 2016, pp. 183–199. DOI: 10.1007/978-3-319-29485-8_11.

[9] G. Hartung, B. Kaidel, A. Koch, J. Koch, and A. Rupp. "Fault-Tolerant Aggregate Signatures". In: *PKC 2016, Part I*. Ed. by C. Cheng, K. Chung, G. Persiano, and B. Yang. Vol. 9614. LNCS. Springer, 2016, pp. 331–356. DOI: 10.1007/978-3-662-49384-7_13.

[10] J. E. Holt. "Logcrypt: forward security and public verification for secure audit logs". In: *AusGrid 2006 and AISW 2006*. Ed. by R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, and W. Susilo. Vol. 54. CRPIT. Australian Computer Society, 2006, pp. 203–211. DOI: 10.1145/1151828.1151852.

[11] Intel Corporation. *2nd Generation Intel Core Mobile Processor Datasheet, Volume 1*. Sept. 2012. URL: https://www-ssl.intel.com/content/www/us/en/processors/core/2nd-gen-core-family-mobile-vol-1-datasheet.html (visited on 05/29/2017).

[12] Intel Corporation. *Intel Core i5-2430M Processor Specification*. URL: https://ark.intel.com/products/53450/Intel-Core-i5-2430M-Processor-3M-Cache-up-to-3_00-GHz (visited on 05/29/2017).

[13] G. Itkis and L. Reyzin. "Forward-Secure Signatures with Optimal Signing and Verifying". In: *CRYPTO 2001*. Ed. by J. Kilian. Vol. 2139. LNCS. Springer, 2001, pp. 332–354. DOI: 10.1007/3-540-44647-8_20.

[14] W. H. Kautz and R. C. Singleton. "Nonrandom binary superimposed codes". In: *IEEE Transactions on Information Theory* 10.4 (1964), pp. 363–377. DOI: 10.1109/TIT.1964.1053689.

[15] H. Krawczyk. "Simple forward-secure signatures from any signature scheme". In: *CCS 2000*. Ed. by D. Gritzalis, S. Jajodia, and P. Samarati. ACM, 2000, pp. 108–115. DOI: 10.1145/352600.352617.

[16] R. Kumar, S. Rajagopalan, and A. Sahai. "Coding Constructions for Blacklisting Problems without Computational Assumptions". In: *CRYPTO 1999*. Ed. by M. J. Wiener. Vol. 1666. LNCS. Springer, 1999, pp. 609–623. DOI: 10.1007/3-540-48405-1_38.

[17] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters. "Sequential Aggregate Signatures, Multisignatures, and Verifiably Encrypted Signatures Without Random Oracles". In: *J. Cryptology* 26.2 (2013), pp. 340–373. DOI: 10.1007/s00145-012-9126-5.

[18] B. Lynn. *The Pairing-Based Crypto Library*. URL: https://crypto.stanford.edu/pbc/ (visited on 05/29/2017).

[19] A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham. "Sequential Aggregate Signatures from Trapdoor Permutations". In: *EUROCRYPT 2004*. Ed. by C. Cachin and J. Camenisch. Vol. 3027. LNCS. Springer, 2004, pp. 74–90. DOI: 10.1007/978-3-540-24676-3_5.

[20] D. Ma. "Practical forward secure sequential aggregate signatures". In: *ASIACCS 2008*. Ed. by M. Abe and V. D. Gligor. ACM, 2008, pp. 341–352. DOI: 10.1145/1368310.1368361.

[21] D. Ma and G. Tsudik. "A New Approach to Secure Logging". In: *Data and Applications Security XXII, IFIP WG 11.3 Working Conference on Data and Applications Security*. Ed. by V. Atluri. Vol. 5094. LNCS. Springer, 2008, pp. 48–63. DOI: 10.1007/978-3-540-70567-3_4.

[22] D. Ma and G. Tsudik. "A new approach to secure logging". In: *ACM Transactions on Storage (TOS)* 5.1 (2009). DOI: 10.1145/1502777.1502779.

[23] D. Ma and G. Tsudik. "Extended Abstract: Forward-Secure Sequential Aggregate Authentication". In: *S&P 2007*. IEEE Computer Society, 2007, pp. 86–91. DOI: 10.1109/SP.2007.18.

[24] B. Schneier and J. Kelsey. "Cryptographic Support for Secure Logs on Untrusted Machines". In: *USENIX 1998, Proceedings*. Ed. by A. D. Rubin. USENIX Association, 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/cryptographic-support-secure-logs-untrusted-machines.

[25] V. Shoup. *NTL: A Library for doing Number Theory*. URL: http://shoup.net/ntl/ (visited on 05/29/2017).