Secure Audit Logs with Verifiable Excerpts – Full Version

Gunnar Hartung

Karlsruhe Institute of Technology, Karlsruhe, Germany
gunnar.hartung@kit.edu

Abstract. Log files are the primary source of information when the past operation of a computing system needs to be determined. Keeping correct and accurate log files is important for after-the-fact forensics, as well as for system administration, maintenance, and auditing. Therefore, a line of research has emerged on how to cryptographically protect the integrity of log files even against intruders who gain control of the logging machine.
We contribute to this line of research by devising a scheme where one can verify integrity not only of the log file as a whole, but also of excerpts. This is helpful in various scenarios, including cloud provider auditing.

Keywords: Secure Audit Logs · Log Files · Excerpts · Forward Security

1 Introduction

Log files are append-only files recording information on events and actions within a computer system. They are essential for digital forensics, intrusion detection and for proving the correct operation of computers.

However, their evidentiary value can be severely impaired if it is unclear whether they have been tampered with. It is therefore imperative to protect log files from unauthorized modification. This need has been widely recognised for a long time, see for example [19, p. 10], [25, Sections 18.3, 18.3.1], [12, Section 8.6].

However, to actually prove a claim e.g. in court with the help of a log file is problematic *even if* the log file's integrity is unharmed, since the log file may contain confidential information that must not be disclosed. Furthermore, a large fraction of log entries may be irrelevant. Filtering these out significantly facilitates the log file analysis.

In this work, we therefore propose a logging scheme that can support the *verification of excerpts* from a log file. Creating an excerpt naturally solves both problems: Log entries that contain confidential and/or irrelevant data can simply be omitted from the excerpt. Excerpts created with our scheme remain verifiable, and therefore retain their probative force. Let us illustrate their use with two examples.

Example 1 (Banking). Consider a bank B that provides financial services to its customers. In order to prove correct behaviour of its computer systems, the bank maintains log files on all transactions on customers' accounts.

When a customer A accuses the bank of fraud or incorrect operation, the bank will want to use its log files to disprove A's allegations. However, submitting the entire log file as evidence to court is not an option, as this would compromise the confidentiality of all transactions recorded, including the ones of other customers. Besides, the log file may also be prohibitively large.

One might alternatively hand the log entries to an independent expert witness, who verifies the log file integrity and then testifies before court on the correct or incorrect operation of the bank. However, this approach eliminates public verifiability, does not solve the problem of the log file size, and still puts the confidentiality of the transactions of all customers at unnecessary risk, even if the expert witness is bound to protect the confidentiality of transactions.

Yet another solution would be to have the entire log file encrypted (under different keys) and to only reveal keys for those log entries that are of interest to the court's proceedings. This would retain the confidentiality of other customers' bank transactions while allowing for public verifiability. But still, this approach does not solve the problem of the log size.

Utilizing a logging scheme with verifiable excerpts, however, the problem at hand is simple: The bank B generates an excerpt from its log files, containing only information on the transactions on A's account and possibly general information, e.g. about the system state. This excerpt is then submitted to court, where it can be verified by the judge and everyone else. If the verification succeeds, the judge may safely consider the information from the excerpt in his/her deliberation.

Example 2 (Cloud Auditing). Imagine an organisation O that would like to use the services of a cloud provider, e.g. for storage. O may be legally required to pass regular audits, and must therefore be able to provide documentation of all relevant events in its computer systems. Therefore, the cloud provider C must be able to provide O with verifiable log files, which can then be included in O's audit report.

Now, if C was to hand over all its log files to O, this would reveal details about other customers' usage of C's services, which would most likely violate confidentiality constraints. Furthermore, once again, the entire log files may be too large for transmission by regular means.

Here, as above, audit logging schemes with verifiable excerpts can solve the problem at hand easily. With these, C could simply create an excerpt containing only information that is relevant for O from its log files. This would solve the confidentiality issue while simultaneously lightening the burden induced by the log file's size, while the excerpt can still be checked by the auditors.

Background. We consider a scenario where there is a single data logger (e.g. a server or a system of multiple servers), who is initially trusted to adhere to a specified protocol, but feared to be corrupted at some point in time. We would like to guarantee that after the logger has been corrupted, it cannot manipulate the log entries created before the corruption.

Preventing the modification of log data usually requires dedicated hardware, such as write-once read-many-times drives (so-called WORM drives) or continuous feed printers. Since employing such hardware may not always be a viable option, cryptographers and computer security researchers have taken on the task to create schemes or protocols to verify the integrity of log files, see e.g. [7], [26], [6], [16], [29], [21], [3], [32], [34]. These schemes cannot protect log data from actual modification, but they can be used to *detect* modifications, while being purely implemented in software. Knowing if and what log data has been tampered with is very valuable information for a forensic investigation.

In order to enable verification, the logger must create a verification key when the logging process is started. This verification key can then be distributed to a set of verifiers, or even published for everyone to see. Since the logger is trusted at the beginning of the process, the verification key is chosen honestly.

In our specific setting, we want the logger to be able to create excerpts from its log files. These excerpts should be verifiable by everyone in possession of the verification key. We demand that it be hard for the adversary to create an excerpt whose content deviates from the information logged honestly while the logger was uncorrupted, yet passes the verification.

Once an attacker has taken control over a system, (s)he may access any cryptographic keys stored within that system, including keys used to create proofs of integrity and authenticity, such as MACs and digital signatures. Using these keys, an attacker can easily forge such proofs, and arbitrarily modify log files without being detected. This renders standard cryptographic schemes useless.

To mitigate this problem, researchers have devised schemes (e.g. [7], [5], [6], [2], [18], [23], [11], [28], [36], [1], [17]) that guarantee "forward integrity" [7]. Such schemes use *a series* of secret keys $sk_0, \ldots, sk_{T-1}$ (instead of a single constant secret key) for authentication and integrity protection, where each key sk_{i+1} can be computed from the previous key sk_i via a specified update procedure. Given $i \in \{0, \ldots, T-1\}$, the verification algorithm then checks whether the data at hand was indeed authenticated using key sk_i. The verification fails if the data has not been authenticated at all or has been authenticated under a different key sk_j with $j \neq i$.

Informally speaking, a scheme has forward integrity if obtaining one of these secret keys sk_i does not help in forging a proof of authenticity and integrity with respect to any previous key sk_j with $j < i$. Digital signature schemes as well as MACs that have forward integrity are also called *forward-secure*.

In this work, we will focus on logging systems that use digital signatures. These have two important advantages over MAC-based logging schemes: Firstly, anyone in possession of the public key pk can verify their integrity, i.e. log files can be verified publicly. Secondly, verifiers can not modify the log file without detection. Due to the symmetric nature of MACs, this *is* possible for MAC-based schemes. On the downside, signature-based logging schemes are usually less efficient than MAC-based schemes.

A secure log file, also called *secure audit log*, can be built from forward-secure signatures schemes as follows [7]. When a new log file is created, the scheme generates a key pair (sk_0, pk). The public key is copied and either published or distributed to a set of verifiers (e.g. auditors). When the logging system is put into operation, log entries are signed with key sk_0, and the resulting signatures are stored along with the log file. At some point in time (for example after a certain amount of time has passed or a certain number of log entries have been signed), the signer updates the secret key sk_0 to sk_1, securely erases[1] sk_0 and continues signing log entries with sk_1 instead of sk_0. At a later point in time, the signer updates sk_1 to sk_2, deletes sk_1 and continues to work with sk_2, and so on. The time interval in which all log entries are signed using the secret key sk_i is called the *i-th epoch*.

When an attacker $\mathcal{A}$ takes control over the system during epoch i (and hence may obtain the secret key sk_i), the forward security of the digital signature scheme or MAC used guarantees that $\mathcal{A}$ cannot modify log entries signed in previous epochs without being detected. Note that $\mathcal{A}$ can trivially forge signatures for the current epoch i and all future epochs by using the regular signing and updating procedures. However, once $\mathcal{A}$ has taken control over the system, (s)he also controls the input to the logging system, and so this cryptographic "weakness" does not give $\mathcal{A}$ more capabilities than it had without the forward-secure signature scheme. When the log file needs to be verified later, everyone who is in possession of pk (or can securely retrieve a copy of it) can run the verification algorithm to see if the log file has been tampered with.

The scheme described above is highly simplified and has several weaknesses. Therefore, actual proposals in the literature as well as current implementations usually employ a combination of the following additional measures.

- Log entries are usually stored together with a timestamp, to detect reordering attacks. [7], [21]
- Many schemes count the number of log entries and add the counter values (*sequence numbers*) to the signatures. This helps determine the order of log entries (that reflect real events in the system) if the log entries do not contain timestamps themselves (or the timestamps have too coarse resolution). [7], [21], [32], [34]
- Some authors (e.g. [26], [21]) have proposed to use *hash chains*, where each log entry is augmented by the hash value the previous log message, which in turn contains the hash value of the previous log message, and so on. This detects reordering attacks as well as deletions of log entries (except from the end of the log file).
- Some schemes add "epoch markers" to the log file to mark an epoch switch. A verifier can then determine which key index i to use for verifying a log entry by counting the number of epoch markers before the log entry. [7]

[1] Erasure of secret keys must be complete and irrecoverable to guarantee security, i.e., the secret keys must actually be overwritten or destroyed, instead of just removing (file) pointers or links to the secret key.

– If a scheme performs epoch switches independently of the amount of time passed since the last epoch switch, it may be sensible to just add a log entry containing the current time in regular intervals. Such log entries are called *metronome entries.* [16]
– Some schemes additionally employ encryption to protect the confidentiality of log messages, e.g. [26], [16]

In our work, we add epoch markers and sequence numbers to log entries. (Event types may also be given by the application.) We abstract from other features. For our purposes, a (plain) log message is just a string of bits $m \in \{0,1\}^*$. This bit string may contain timestamps and/or event types, may be formatted in any fashion and may be encrypted or not. Log messages may also be categorized (e.g. by the event type), in which case they contain a set N of category names that the log message belongs to. Our scheme supports log entries belonging to any number of categories. (See section 4 for more details.) We focus on the secure *storage* of log entries, instead of also considering the secure *transmission* of log entries to a logging server, since this problem is mostly orthogonal to the storage problem.

Previous and Related Work. The oldest mentioning of protocols to protect the integrity of log files appears to be due to Futoransky and Kargieman [14,15], but passed mostly unnoticed.

The study of cryptographic mechanisms to protect log files has been brought to wider attention by Bellare and Yee [7] in 1997. Motivated by the task to verify the operation of an initially trusted machine in an untrusted and potentially adversarial environment, they introduced the notion of forward integrity for MAC schemes. Intuitively, this notion requires that, if the trusted machine is corrupted at some point in time T_c, but uncorrupted before that point in time, then all modifications of log entries added (sufficiently long) before T_c can be detected with very high probability.

Bellare and Yee developed a simple scheme of forward-secure MACs (based on a key-chain generated by a pseudorandom function) and augmented that scheme with sequence numbers and epoch markers to add protection against the deletion of individual log entries.

Schneier and Kelsey [26,27] devised a more concrete scheme for secure logging using MACs. (The MAC key is continuously evolved using a hash function, similar to Bellare and Yee's scheme.) Schneier and Kelsey assume an untrusted machine U collecting the log entries, a trusted machine T that holds the initial MAC key (and thus can verify the complete log) and a semi-trusted log verifier V. Their scheme includes encryption of log entries and a mechanism for T to grant the semi-trusted verifier V read access to individual log entries.

Building on their scheme, Holt [16] designed Logcrypt. Holt used a construction similar to the Schneier-Kelsey scheme, but proposed to substitute digital signatures for the MACs used by Schneier and Kelsey. While this change decreases performance, it allows for publicly verifiable log files, since the verification key can be made public. Public verifiability may be an essential feature in some applications, such as cryptographic voting schemes.

Marson and Poettering [24] devised "Seekable Sequential Key Generators" for a secure logging scenario (using MACs). These "SSKGs" basically form a hash chain based on a one-way function, where one can efficiently "seek forward", i.e. given the i-th element in the chain, one can quickly compute the n-th element for each $n \geq i$ without having to evaluate the one-way function $n - i$ times.

Ma and Tsudik [21,22] have shown that Schneier's and Kelsey's semi-trusted verifier V can easily be tricked into accepting a modified log file. This was termed a "delayed detection attack", since the fully trusted verifier T can indeed detect such tampering, but is considered to check the log file at a later point in time. Moreover, Ma and Tsudik showed a "truncation attack" on the previous schemes, where the attacker deletes one or more log entries from the tail of the log file. (This truncation attack also applies to Logcrypt, which was already acknowledged in [16]. Holt proposed to use metronome entries to deal with this issue.)

In response to these attacks, Ma and Tsudik devised "forward secure sequential aggregate" signatures (FssAgg signatures). These are closely related to aggregatable signatures (like the B(G)LS scheme [9,10,8]), but impose an order on the set of aggregated messages by requiring that each message shall be signed together with a counter.[2] In order to achieve forward security, they combine several instances of the B(G)LS scheme, where the secret keys are not chosen independently, but the secret key for each epoch is the hash value of the secret key for the previous epoch. (The hash function needs to be modelled as a random oracle to allow for provable security.) However, the public key size of their scheme is linear in the number of epochs T.

Later on, Ma [20] devised further FssAgg signature schemes that offer different tradeoffs in efficiency and build on other hardness assumptions than the B(G)LS scheme.

Since FssAgg schemes are public-key primitives, the verification key can be given to any verifier, preventing delayed detection attacks. Moreover, since only one (aggregated) signature needs to be kept in order to verify the log file, truncation attacks can be detected, as long as the attacker cannot "deaggregate" signatures for individual log entries from the aggregate signature.

While providing a single aggregate signature for the complete log file averts truncation attacks, it also eliminates the possibility to check the integrity of individual log entries without checking the entire log file. In order to re-enable the verifier to do so, Ma and Tsudik modified their scheme to include an individual signature for each log entry as well as an aggregated signature for all log entries. This forced them to reconsider the deaggregation problem and strengthen

[2] The term "sequential aggregate signatures" is also used to denote aggregate signature schemes where aggregation is not a public, ad-hoc operation (where given any two sets M_0, M_1 of messages and the corresponding signatures σ_0, σ_1 it is possible to derive σ for $M := M_0 \cup M_1$), but where only the signer of a message m can create an aggregated signature for $M := M_0 \cup \{m\}$.

their security notion to so-called "*immutable* forward-secure sequential aggregate signatures".

Driven by performance considerations on the signer side, Yavuz, Peng and Reiter [32,33] designed a scheme called "Blind-Aggregate-Forward" (BAF). While BAF has a very efficient signing procedure, the size of the public verification key is linear in the maximum number of supported epochs. While this is a sensible trade-off for applications where signers are subject to tight resource constraints (such as wireless sensors), it may be undesirable in other applications.

Another scheme by Yavuz, Peng and Reiter is LogFAS [34,35]. The verification algorithm for LogFAS requires less computational effort than BAF's verification algorithm, but the sizes of signing *and* verification keys for LogFAS are linear in the number of supported log entries. This might make LogFAS a reasonable choice for applications where the signer needs to generate signatures quickly, but has sufficient storage (e.g. a server facing a high load).

Waters et al. [30] focus on encryption of log entries in a way that allows for efficient keyword-search in the log file. They do not develop new techniques to guarantee log file integrity, or to guarantee the integrity of the "excerpt" of the log file that is returned by a keyword-search. Therefore, their contribution is orthogonal to ours. (In fact, combining their scheme with ours would be very interesting.)

Stathopoulos et al. [29] take a management point of view on secure logging. They build upon the Schneier and Kelsey scheme and add another trusted authority which is given signatures of the current log file state at regular intervals. This gives an additional way of detecting modifications to log files.

Wensheng et al. [31] build a web service for secure audit logs. They also build on Schneier's and Kelsey's scheme, but use the Trusted Computing Base to store cryptographic keys.

The notion of excerpts from log files has not been explicitly considered before. We note, though, that LogFAS [34,35] can support the verification of *arbitrary* subsequences of log files. However, this is more an accidental property of the LogFAS construction than due to an explicit design goal, and furthermore, systems that can verify *every* subsequence are in general not suited for our example applications, as will be discussed in Section 3.

Closest to our work is the scheme by Crosby and Wallach [13], who devised a method for secure logging that allows for controlled deletion of certain log entries while keeping the remaining log entries verifiable. However, their scheme relies on frequent communication between the log server and one or more trusted auditors that need to store "commitments" to the log file, whereas our scheme can be used non-interactively. Furthermore, they did not formulate a security notion and consequently did not give a proof of security for their scheme.

Finally, we point out a survey paper on secure logging by Accorsi [3], which gives an overview on some of the older schemes mentioned above.

Our Contribution. Our contribution is twofold: Firstly, we develop a model for secure logging with verifiable excerpts. The ability to verify excerpts can be useful (i) to provide full confidentiality and privacy of most of the log entries, even when a subset of the log entries needs to be disclosed, (ii) to save resources during transmission and storage of the excerpt, and (iii) to ease manual review of log files. We also develop a strong, formal security notion for such schemes.

Secondly, we propose a novel audit logging scheme that allows for verification of excerpts. Our scheme may be used to verify both the *correctness* of all log entries contained in an excerpt as well as the *completeness* of the excerpt, i.e. the presence of all relevant log entries in the excerpt. We rely on the application software to define which log entries are relevant for the excerpts. Our scheme makes efficient use of a forward-secure signature scheme, which is used in a black-box fashion. Therefore, our scheme can be instantiated with an arbitrary forward-secure signature scheme and thereby tuned to meet specific performance goals, and be based on a wide variety of hardness assumptions. We analyse our scheme formally and give a perfectly tight reduction to the security of the underlying forward-secure signature scheme.

Outline. Section 2 introduces preliminary definitions and some notation. In Section 3, we develop a formal framework to reason about log files with excerpts, and give a security definition for such schemes. Section 4 presents our construction, proves that it fulfills the security notion from Section 3, and analyses the overhead imposed by our scheme. It also compares our scheme to other schemes from the literature. Finally, Section 5 concludes the paper.

2 Preliminaries, Notation and Conventions

Sequences. Let $S = \langle s_0, \ldots, s_{l-1} \rangle = \langle s_i \rangle_{i=0}^{l-1}$ be a finite (possibly empty) sequence over some domain D. Then $|S| := l \in \mathbb{N}_0$ denotes the *length* of S. We write $v \in S$ to indicate that v is contained in S, i.e., there exists an $i \in \{0, \ldots, l-1\}$ such that $v = s_i$. The empty sequence is $\langle\rangle$. The concatenation of two finite sequences S_1, S_2 is denoted as $S_1 \,\|\, S_2$. If $s \in D$ is a single element, we write $S_1 \,\|\, s$ as a shorthand for $S_1 \,\|\langle s \rangle$. If $S = \langle s_0, \ldots, s_{l-1} \rangle$ is a sequence of length $l \in \mathbb{N}_0$ and $P = \langle s_0, \ldots, s_{m-1} \rangle$ for some $m \leq l$, then P is a *prefix* of S. If $I := \langle i_0, \ldots, i_{n-1} \rangle$ is a (possibly empty) finite, strictly increasing sequence of numbers $i_j \in \{0, \ldots, l-1\}$ (for all $j \in \{0, \ldots, n-1\}$, with $n \in \mathbb{N}_0, n < l$), we call I an *index sequence for* S and $S' = \langle s_{i_0}, \ldots, s_{i_{n-1}} \rangle$ the *subsequence of* S *induced by* I.

Definition 1 (Operations on Subsequences). *Let* $S = \langle s_0, \ldots, s_{l-1} \rangle$*; let* $I = \langle i_0, \ldots, i_{v-1} \rangle$*,* $J = \langle j_0, \ldots, j_{w-1} \rangle$ *be two index sequences for* S*, and let* $T = \langle s_{i_0}, \ldots, s_{i_{v-1}} \rangle$*,* $U = \langle s_{j_0}, \ldots, s_{j_{w-1}} \rangle$ *be the subsequences of* S *induced by* I *and* J*, respectively. Then:*

$T \cup U$

is the subsequence of S *that contains exactly those elements* s_k *for which* $k \in I$ *or* $k \in J$ *or both, in the order of increasing* $k \in \{0, \ldots, l-1\}$*,*

$T \cap U$

is the subsequence of S that contains exactly those elements s_k for which $k \in I$ and $k \in J$, in the order of increasing $k \in \{0, \ldots, l-1\}$.

Note that if S contains duplicates, then there may be different index sequences inducing the same subsequence. Therefore, the operations from Definition 1 are only well-defined if the index sequences I and J are given. In this work, we will omit specifying I and J when they are clear from the context.

Example 3. Let $S = \langle s_0, \ldots, s_5 \rangle$, and let $I := \langle 0, 3, 5 \rangle$, $J := \langle 2, 3, 4 \rangle$ define the subsequences T and U of S. Then we have $T \cup U = \langle s_0, s_2, s_3, s_4, s_5 \rangle$ and $T \cap U = \langle s_3 \rangle$. Note that even if, e.g. $s_4 = s_5$, we would still have $T \cap U = \langle s_3 \rangle$, since the operations are defined based in the indices i of the elements s_i in the sequence S, not based on the equality in the domain D.

General Notation. A log entry m is a bit string, i.e. $m \in \{0,1\}^*$. Log entries are also called log messages or just messages. The concatenation operation on bit strings is also denoted by $\|$, just as the concatenation of sequences. A log file $M = \langle m_0, \ldots, m_{l-1} \rangle$ is a finite, possibly empty sequence of log entries.[3]

We write $X := V$ for a deterministic assignment operation. In contrast, $X \leftarrow V$ is used when V is a finite set and X is chosen uniformly at random from V, or V is a probabilistic algorithm and X is assigned the output of that algorithm. All random choices are considered to be independent. We write PPT for "probabilistic polynomial time". Throughout this paper, $\kappa \in \mathbb{N}_0$ is the security parameter. All algorithms are implicitly given 1^κ as an additional input. The set of all polynomials $p : \mathbb{N}_0 \to \mathbb{N}_0$ which are parameterized by κ is $\mathrm{poly}(\kappa)$.

A function $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ is called *negligible* iff for each constant $c \in \mathbb{N}$ there exists a number $n_c \in \mathbb{N}$ such that $f(n) \leq n^{-c}$ for all $n \geq n_c$. We write $f(n) \leq \mathrm{negl}(n)$ if so. A function $g : \mathbb{N} \to [0,1]$ is called *overwhelming* if $g'(n) := 1 - g(n)$ is negligible.

Forward-Secure Signature Schemes.

Definition 2 (Key-Evolving Signature Scheme, based on [5]). *A key-evolving digital signature scheme $\Sigma = (\mathrm{KeyGen}, \mathrm{Update}, \mathrm{Sign}, \mathrm{Verify})$ is a tuple of PPT algorithms, which are described as follows.*

$\mathrm{KeyGen}(T)$

receives an a priori upper bound T on the number of epochs as input. It generates and outputs a pair of keys, consisting of the initial private signing key sk_0 and the public verification key pk.

$\mathrm{Update}(sk_i)$

takes a secret key sk_i as input, evolves it to sk_{i+1} and outputs sk_{i+1}. The old signing key sk_i is then deleted in an unrecoverable fashion. If $i \geq T-1$, the behaviour of Update *may be undefined.*

[3] Note that $M = \langle m_0, \ldots, m_{l-1} \rangle \neq m_0 \| \ldots \| m_{l-1}$, i.e. we consider the log entries in M to be distinguishable.

$\mathrm{Sign}(sk_i, m)$
: *computes and outputs a signature σ for a given message $m \in \{0,1\}^*$, using a secret key sk_i.*

$\mathrm{Verify}(pk, m, i, \sigma)$
: *checks if σ is a valid signature under public key pk, created with the i-th secret key, for a given message m. If it deems the signature valid, it outputs 1, otherwise it outputs 0.*

We require correctness *in the sense that for each security parameter $\kappa \in \mathbb{N}_0$, for each polynomial bound $T := T(\kappa) \in \mathrm{poly}(\kappa)$ on the number of epochs, for each index $i \in \{0, \ldots, T-1\}$, and each message $m \in \{0,1\}^*$ the following equation holds with overwhelming probability:*

$$\mathrm{Verify}(pk, m, i, \mathrm{Sign}(sk_i, m)) = 1 \ ,$$

where $(sk_0, pk) \leftarrow \mathrm{KeyGen}(T)$, and $sk_i = \mathrm{Update}^i(sk_0)$, i.e. sk_i is the initial secret key sk_0 updated i times. The probability is measured over the randomness used by the algorithms KeyGen*,* Update*,* Sign *and* Verify *(if any).*

We assume without loss of generality that the message space of each signature scheme is $\{0,1\}^*$. If a signature scheme only supports a signature space $\mathcal{M} \neq \{0,1\}^*$, we assume the presence of a collision resistant hash function $H : \{0,1\}^* \to \mathcal{M}$. We also assume that the algorithms Update and Sign have access to the public key and that the index i of a secret key sk_i can be extracted from sk_i efficiently.

The security notion for key-evolving signature schemes is mostly similar to the standard notion of *existential unforgeability under chosen message attacks*, but slightly more complicated, due to the presence of different epochs. It captures the "forward security" property.

Definition 3 (Forward-Secure Existential Unforgeability under Chosen Message Attacks). *The notion of forward-secure existential unforgeability under chosen message attacks is defined based on an experiment parameterized by a key-evolving signature scheme $\Sigma = (\mathrm{KeyGen}, \mathrm{Update}, \mathrm{Sign}, \mathrm{Verify})$, a PPT adversary $\mathcal{A}$, the number of epochs $T := T(\kappa) \in \mathrm{poly}(\kappa)$ and the security parameter κ.*

Setup Phase.
: *The experiment begins by creating a pair of keys $(sk_0, pk) \leftarrow \mathrm{KeyGen}(T)$, and initializing a counter $i := 0$. Afterwards $\mathcal{A}$ is called with inputs pk and T.*

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

Signature Oracle.
: *On input $m \in \{0,1\}^*$, the signature oracle computes the signature $\sigma = \mathrm{Sign}(sk_i, m)$ for m using the current secret key sk_i. It returns σ to $\mathcal{A}$.*

Epoch Switching Oracle.

Whenever $\mathcal{A}$ triggers the NextEpoch *oracle, the experiment sets $sk_{i+1} \leftarrow$ Update(sk_i) and $i := i + 1$. The oracle returns the string "OK" to the adversary. $\mathcal{A}$ may invoke this oracle at most $T - 1$ times.*

Break In.

Once in the experiment, the attacker may query a special BreakIn *oracle that stores the current epoch number as $i_{\text{BreakIn}} := i$ and returns the current secret key sk_i to the adversary. After $\mathcal{A}$ has invoked this oracle, it is no longer allowed any oracle queries (neither to the* BreakIn *oracle, nor to its other oracles).*[4]

Forgery Phase.

Finally, the attacker outputs a forgery (m^*, i^*, σ^*). The experiment outputs 1 *iff* Verify$(pk, m^*, i^*, \sigma^*) = 1$, *$m^*$ was not submitted to the signature oracle during epoch i^*, and $i^* < i_{\text{BreakIn}}$. (Let $i_{\text{BreakIn}} := \infty$ if $\mathcal{A}$ did not use its* BreakIn *oracle.) If any of these conditions is not met, the experiment outputs* 0.

We say that $\mathcal{A}$ wins *an instance of this experiment iff the experiment outputs* 1. *A key-evolving signature scheme $\Sigma =$* (KeyGen, Update, Sign, Verify) *is said to be* forward-secure existentially unforgeable under chosen message attacks (or FS-EUF-CMA-secure) *if for each PPT adversary $\mathcal{A}$ and each $T \in \text{poly}(\kappa)$ the above experiment outputs* 1 *with only negligible probability (in κ).*

3 Secure Logging with Verifiable Excerpts

We now develop a formal model for log files with excerpts. Obviously, given a log file M, an excerpt E is a subsequence of M. However, a scheme where *each* subsequence of M can be verified[5] is not sufficient for our applications, since the provider of the excerpt could simply omit some critical log entries. Put differently, such a scheme may guarantee correctness of all log entries in the excerpt, but it does not guarantee that all relevant log entries are present.

To address this problem, we introduce *categories*. Each log entry is assigned to one or more categories, which may also overlap. Each category has a unique name $\nu \in \{0,1\}^*$. We require that when a new log entry m is appended to the log file, one must also specify the names of all categories that m is assigned to.

We return to our banking example from Section 1 to illustrate the use of such categories. The bank B introduces a category C_A for each customer A, and then adds each log entry concerning A's account to C_A. The problem of checking the completeness of the excerpt for A's account is thereby reduced to checking the presence of all log entries from the category C_A and possibly

[4] This restriction is without loss of generality, since the adversary knows $sk_{i_{\text{BreakIn}}}$ after this query and can thus create signatures as well as all subsequent secret keys by itself. Also, triggering the NextEpoch oracle after the BreakIn oracle would have no consequences on the outcome of the game.

[5] LogFAS [34,35] offers such a capability.

from other categories containing general information. Of course, categories may also be added based on other criteria, such as the event type (e.g. creation and termination of an account, deposition or withdrawal of funds, and many more). Note that the set of categories is not fixed in advance; rather the bank must be able to add new categories on-the-fly, as it gains new customers. The use of categories is similar in the cloud provider example.

3.1 Categorized Logging Schemes

Definition 4 (Categorized Messages and Log Files). *A categorized message (also categorized log entry) $m = (N, m')$ is a pair of a finite, non-empty set N[6] of category names $\nu \in \{0,1\}^*$ and a log entry $m' \in \{0,1\}^*$. A categorized log file $M = \langle m_0, \ldots, m_{l-1} \rangle$ is a finite, possibly empty sequence of categorized log entries m.*

When it is clear from the context that we mean categorized log entries or categorized log files, we will omit the term "categorized" for the sake of brevity. In particular, this section as well as the following one will mainly be concerned with categorized log entries and categorized log files.

Definition 5 (Categories). *A* category *with name $\nu \in \{0,1\}^*$ of a categorized log file $M = \langle (N_i, m'_i) \rangle_{i=0}^{l-1}$ is the (possibly empty) subsequence C of M that contains exactly those log entries $(N_i, m'_i) \in M$ where $\nu \in N_i$. C is denoted by $C(\nu, M)$. C's* index sequence *$I(\nu, M)$ is the (possibly empty, strictly increasing) sequence that contains all $i \in \{0, \ldots, l-1\}$ for which $\nu \in N_i$.*

Definition 6 (Excerpts). *Given a categorized log file $M = \langle m_i \rangle_{i=0}^{l-1}$ and a finite set N of category names, the* excerpt for *N is $E(N, M) = \bigcup_{\nu \in N} C(\nu, M)$. The* index sequence *$I(N, M)$ is the (possibly empty, strictly increasing) sequence of all i with $i \in I(\nu, M)$ for at least one $\nu \in N$.*

Clearly, $C(\nu, M)$ is induced by $I(\nu, M)$, and $E(N, M)$ is induced by $I(N, M)$. In the following, we will mostly omit the second parameter, since it will be clear from the context. Moreover, we make the convention that there is a category named "All" such that $C(\mathrm{All}) = M$, i.e. $\mathrm{All} \in N_0 \cap \ldots \cap N_{l-1}$. As a special case of excerpts, we obtain M as an excerpt for the categories $N = \{\mathrm{All}\}$.

In the following, we adopt the convention that variables with two indices are an "aggregate" of values ranging from the first to the second index, i.e. $\sigma_{0,j}$ is the aggregate of $\sigma_0, \ldots, \sigma_j$. In our case, this aggregate is simply a sequence of the individual values, i.e. $\sigma_{0,j} := \langle \sigma_0, \ldots, \sigma_j \rangle$, $M_{0,j} := \langle m_0, \ldots, m_j \rangle$. However, $\sigma_{0,j}$ may in general also be an actual aggregate signature, as in [21].

Definition 7 (Categorized Key-Evolving Audit Log Scheme). *A categorized key-evolving audit log scheme is a quintuple of probabilistic polynomial time algorithms $\Sigma = (\mathrm{KeyGen}, \mathrm{Update}, \mathrm{Extract}, \mathrm{AppendAndSign}, \mathrm{Verify})$, where:*

[6] This is intended as the upper case greek letter ν, which unfortunately looks identical to the upper case latin letter n.

KeyGen(T)

outputs an initial signing key sk_0, a permanent verification key pk, and an initial signature $\sigma_{0,-1}$ for the empty log file. T is the number of supported epochs.

Update(sk_i, M, σ)

evolves the secret key sk_i for epoch i to the subsequent signing key sk_{i+1} and then outputs sk_{i+1}. sk_i is erased securely. Update *may also use and modify the current log file M as well as the current signature σ, e.g. by adding epoch markers or metronome entries.*

Extract($sk_i, M_{0,j-1}, \sigma_{0,j-1}, N$)

takes a log file $M_{0,j-1}$ together with a signature $\sigma_{0,j-1}$ for $M_{0,j-1}$ and a set N of category names and outputs a signature σ for the excerpt $E(N) = E(N, M_{0,j-1})$, computed with the help of sk_i.

AppendAndSign($sk_i, M_{0,j-1}, m_j, \sigma_{0,j-1}$)

takes as input the secret key sk_i, the current log file $M_{0,j-1}$, its signature $\sigma_{0,j-1}$ and a new log entry m_j and outputs a signature $\sigma_{0,j}$ for $M_{0,j} := M_{0,j-1} \,\|\, m_j$.

Verify(pk, N, E, σ)

is given the verification key pk, a set $N = \{\nu_0, \ldots, \nu_{n-1}\}$ of category names, an excerpt E and a signature σ. It outputs 1 or 0, where 1 means $E = E(N, M)$, and 0 means $E \neq E(N, M)$. Again, by choosing $N = \{\mathrm{All}\}$, one can verify the entire log file up until epoch i.

We require correctness in the following sense: For each $\kappa \in \mathbb{N}_0$, $T = T(\kappa) \in \mathrm{poly}(\kappa)$, $l = l(\kappa) \in \mathrm{poly}(\kappa)$, each sequence $M_{0,l} = \langle m_0, \ldots, m_l \rangle$ of categorized log entries, each increasing sequence $I = \langle i_0, \ldots, i_l \rangle$ with $i_j \in \{0, \ldots, T-1\}$, for each set of category names N, and for pk, σ created by the process described below, we have that:

$$\Pr\left[\mathrm{Verify}(pk, N, E(N, M_{0,l}), \sigma) = 1\right] \text{ is overwhelming in } \kappa,$$

where the probability is measured over the coins used by Verify *(if any) and the coins used by* KeyGen, Update, AppendAndSign *and* Extract *in the process below. The process for creating pk and σ is as follows:*

1. *Let $(sk_0, pk, \sigma_{0,-1}) \leftarrow \mathrm{KeyGen}(T)$, $i := 0$, $M_{0,-1} := \langle\rangle$, and $\sigma := \sigma_{0,-1}$.*
2. *Iterate over all $j \in \{0, \ldots, l\}$ in increasing order:*
   (a) While $i_j > i$, compute $sk_{i+1} \leftarrow \mathrm{Update}(sk_i, M_{0,j-1}, \sigma)$ and set $i := i+1$.
   (b) Set $\sigma \leftarrow \mathrm{AppendAndSign}(sk_{i_j}, M_{0,j-1}, m_j, \sigma)$.
   (c) Set $M_{0,j} := M_{0,j-1} \,\|\, m_j$.
3. *Output pk and $\sigma \leftarrow \mathrm{Extract}(sk_{i_l}, M_{0,l}, \sigma, N)$.*

The process used for the definition of correctness models regular usage of Σ. Here, the m_j are the log entries to be added, and each i_j corresponds to the epoch during which m_j is added to the log file.

Note that we require Verify to validate excerpts without actually “knowing” the complete log file. This is the main difficulty that our construction must overcome.

3.2 General Remarks

Remark 1 (Reset Attacks). It is quite obvious that once an attacker has seen a valid signature σ for a log file M from some point in time t_0, (s)he can reset the entire log file to M and restore the previous signature σ once (s)he has control over the log server. Since one requires that $\text{Verify}(pk, \{\text{All}\}, M, \sigma) = 1$ at t_0, we cannot expect $\text{Verify}(pk, \{\text{All}\}, M, \sigma) = 0$ at some later point in time t_1, unless Verify has an additional trusted input such as the current time or the number of messages that have been added to the log file so far.

But even if Verify has such a trusted input, it is questionable whether one wants excerpts to become invalid over time, and if so after what amount of time. This appears to be an aspect that depends heavily on the envisaged application.

We therefore take a different path and let excerpts remain (cryptographically) valid for an indefinite amount of time. It is then up to the application to decide whether an excerpt is "fresh enough". This is sufficient for both our examples, where only an a posteriori verification of events is required, and everyone can see whether an excerpt spans the time period of interest.

Remark 2 (Secret Keys for Generation of Excerpts). In our model, creating an excerpt from a log file M and a corresponding signature σ requires a secret key. This is a helpful measure against adversaries that do not get to know a secret key, but does not offer protection against adversaries that *do* obtain a secret key (using their BreakIn oracle).

Consider our model, where the secret key may be used in the extraction algorithm. In a sane design, this secret key may only be used to authenticate some information by signing it. (Using a secret signing key for anything else than signing a message would violate sensible and well-established design principles.) Now suppose that an excerpt is generated in epoch i, but the last log entry to be included in the excerpt was added in epoch $j < i$. Now, since sk_j has been deleted already, the only secret key available in epoch i is sk_i. So whatever information is signed during the extraction process can only be signed under sk_i.

However, by then, the attacker may already have broken into the server and stolen the secret key sk_i. Now the adversary may use this secret key to sign any false claim during the extraction algorithm. This information will be accepted by the verification algorithm, since it has a valid signature.

Thus, even if some information is authenticated with a signing key in the extraction process, that information can not be trusted to be true, if one considers an adversary that obtains a secret key at some point in time. Then, however, there is no need to sign it in the first place, and no need to use a signing key in the extraction procedure.

While the discussion above is highly informal, we believe it plausibly demonstrates that "adding new signatures" during the extraction does not offer any increased security against adversaries that obtain a secret key.

Our reason for still using the secret key during extraction is the added protection against attackers that do not obtain the secret key. If one requires the entire excerpt to be signed together with a timestamp and the set of categories being requested, then an adversary trying to create a signature for *any* excerpt must forge a new signature, which is very hard without the secret key.

3.3 Security Model

We now define our security notion for categorized key-evolving audit log schemes. It is similar to the above definition for key-evolving signature schemes, but adjusted to the append-only setting and to support extraction queries by the attacker.

Definition 8 (Forward-Secure Existential Unforgeability under Chosen Log Message Attacks). *For a categorized key-evolving audit log scheme* $\Sigma = (\mathrm{KeyGen}, \mathrm{Update}, \mathrm{Extract}, \mathrm{AppendAndSign}, \mathrm{Verify})$*, a PPT adversary* $\mathcal{A}$*, the number of epochs* $T := T(\kappa) \in \mathrm{poly}(\kappa)$ *and the security parameter* $\kappa \in \mathbb{N}_0$*, the security experiment FS-EUF-CLMA-Exp*$_{\Sigma,\mathcal{A},T}(\kappa)$ *is defined as follows:*

Setup Phase.
The experiment generates the initial secret key, the public key and the initial signature as $(sk_0, pk, \sigma_{0,-1}) \leftarrow \mathrm{KeyGen}(T)$*. It initializes the epoch counter* $i := 0$*, the message counter* $j := 0$*, and the log file* $M_{0,-1} := \langle\rangle$*. It then starts* $\mathcal{A}$ *with inputs* pk*,* T *and* $\sigma_{0,-1}$*.*

Query Phase.
During the query phase, the adversary may adaptively issue queries to the following four oracles:

Signature Oracle.
Whenever $\mathcal{A}$ *submits a message* m_j *to the signature oracle, the experiment appends that message to the log file by setting* $M_{0,j} := M_{0,j-1} \,\|\, m_j$ *and updates the signature to*

$$\sigma_{0,j} \leftarrow \mathrm{AppendAndSign}(sk_i, M_{0,j-1}, m_j, \sigma_{0,j-1}) \ .$$

It then sets $j := j + 1$*. The oracle returns the new signature* $\sigma_{0,j}$*.*

Extraction Oracle.
On input of a set N *of category names, the experiment creates a signature* $\sigma \leftarrow \mathrm{Extract}(sk_i, M_{0,j-1}, \sigma_{0,j-1}, N)$ *for the excerpt* $E := E(N, M_{0,j-1})$ *and gives* (E, σ) *to the adversary.*

Epoch Switching Oracle.
Upon a query to the NextEpoch *oracle, the experiment moves to the next epoch, updating the secret key (and possibly the log file and its signature) to* $sk_{i+1} \leftarrow \mathrm{Update}(sk_i, M_{0,j-1}, \sigma_{0,j-1})$ *and incrementing the epoch counter* $i := i + 1$*. The oracle returns the updated log file* M' *and signature* σ' *to the attacker. This oracle may be queried at most* $T - 1$ *times.*

Break In.
Optionally, the adversary may use its BreakIn *oracle to retrieve the current secret key* sk_i*. After this, it may no longer issue queries to any of*

its oracles.[7] *The experiment sets* $i_{\text{BreakIn}} := i$. *(Let* $i_{\text{BreakIn}} := \infty$ *if* $\mathcal{A}$ *never queried this oracle.)*

Forgery Phase.

At the end of the experiment, $\mathcal{A}$ *outputs a non-empty set* N^* *of categories, a forged excerpt* E^* *for* N^**, and a forged signature* σ^* *of* E^**.*

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

- *The signature is* valid*, i.e.* $\text{Verify}(pk, N^*, E^*, \sigma^*) = 1$.
- *The signature is* non-trivial*, i.e. it meets the following requirements:*
  - E^* *has not been part of an answer of the extraction oracle to* $\mathcal{A}$ *for the categories* N^**. More formally, if* $N_0, \ldots, N_k$ *are the sets of category names that* $\mathcal{A}$ *used to call its extraction oracle and* $E_0, \ldots, E_k$ *are the excerpts returned by the oracle, then we require* $(N^*, E^*) \notin \{(N_0, E_0), \ldots, (N_k, E_k)\}$.
  - *If* $\mathcal{A}$ *used its* BreakIn *oracle to obtain a secret key* sk_i*, let* $E_i = E(N^*, M_i)$*, where* M_i *is the log file at the time of switching from epoch* $i_{\text{BreakIn}} - 1$ *to epoch* i_{BreakIn}*. (Formally,* M_i *is the log file returned by the most recent call to the* NextEpoch *oracle, so* M_i *includes all changes made by the* Update *algorithm. We let* $M_i := \langle\rangle$ *if* $\mathcal{A}$ *never called the* NextEpoch *oracle.) We require that* E_i *is not a prefix of* E^**. Put differently,* E^* *must not just be a continuation/extension of* E_i*.*

We say that $\mathcal{A}$ *lost the experiment, iff* $\mathcal{A}$ *did not win the experiment. A categorized key-evolving audit log scheme* Σ *is said to be* FS-EUF-CLMA-secure*, iff for all* $T = T(\kappa) \in \text{poly}(\kappa)$ *and all probabilistic polynomial time attackers* $\mathcal{A}$ *the probability for* $\mathcal{A}$ *winning the above experiment is negligible in* κ*.*

Let us review the above definition. As for standard security notions, we let the adversary completely determine the input to the cryptographic scheme, except for the keys. In our case, this input consists of the messages being submitted to the log (using the signature oracle) as well as the *timing* of these messages (controlled by the order in which $\mathcal{A}$ submits these to the signing oracle as well as the NextEpoch oracle). While such a powerful adversary may be unrealistic in most real-world scenarios, giving the adversary such power in the experiment results in a stronger security notion. We only allow the attacker to move forward in time, i.e., we assume the attacker does not have a time machine.

Moreover, we grant the adversary access to any signature that is created during the experiment by returning the signature created by the signature queries, as well as the updated signatures created during epoch switches. Furthermore, the adversary may explicitly request a signature for any excerpt. This models a scenario where the attacker might learn signatures from court proceedings, where the bank needs to prove its correct behaviour.

The adversary wins the experiment if it manages to output a forged signature σ^* together with a forged excerpt E^* for any categories N^* of its choice. We want to exclude trivial wins from our definition, and therefore require that E^* was

[7] Again, this restriction is without loss of generality, see footnote 4 on page 11.

never requested by $\mathcal{A}$ as an excerpt for the categories N^*. Again, this is similar to standard security notions.

Furthermore, we must add an additional restriction if $\mathcal{A}$ obtained a secret key sk_i. We require that E^* is not simply an extension of the "real" excerpt E_i up until the end of epoch $i-1$, or, stating this the other way round, that E_i is not a prefix of the forged excerpt E^*. This restriction is necessary, since creating such extensions is trivial, given the secret key sk_i. The adversary simply needs to run the algorithms AppendAndSign and Extract (and possibly Update) of Σ, given the signature σ_i from the epoch switch to epoch i (returned to $\mathcal{A}$ by the NextEpoch oracle) and the secret key sk_i.

Observe that our security model allows a log file to be truncated to the state of the most recent epoch switch, counting this as a trivial attack. As explained in Remark 1 on page 14 such attacks are always possible.

We acknowledge this is a weakness of our model, but argue that it is a common one. We do not know of schemes that actually offer protection against such attacks, except the [21,22] scheme where log entries can not be individually verified. (Ma and Tsudik [21,22] also propose schemes that offer individual verification. These schemes, however, only offer protection against attackers that try to truncate the log file to a state before the most recent "anchor point". The epoch markers of our scheme can be viewed as such "anchor points".) Thus, our model does not stand back when compared to previous work.

It is an open question to develop a scheme where log entries can be verified individually and *all* truncation attacks are hard to perform. This question is subject to ongoing research.

4 Our Scheme

We now describe a scheme that realizes the above security notion. We call it SALVE, for "Secure Audit Log with Verifiable Excerpts". The main ingredient for SALVE[8] is a forward-secure signature scheme. Let us briefly describe the basic ideas underlying our construction.

Sequence Numbers per Category.
Instead of adding only global sequence numbers, we augment signatures with sequence numbers (counters) c_ν for *each* category ν. In particular, the sequence numbers for the category All work as global sequence numbers.

Signing Counters.
Each log entry is signed along with the sequence numbers belonging to the categories of the log entry. All these counters are increased by one after the log entry has been signed. During verification, one checks if the counters of each category ν supposed to be present in the excerpt form the sequence $\langle 0, \ldots, c_\nu - 1\rangle$. This way, one can detect duplicate log entries, log entries missing between present ones, and reordering attacks.

[8] "This is what passes for humor amongst cryptographers." [4]

Epoch Markers with Counters.

Additionally, we sign all counters that have changed during an epoch i together with the epoch markers created at the end of epoch i. These epoch markers are signed using the secret key, which is then evolved using the Update algorithm. This provides protection against truncation attacks that try to truncate the log file to a state before the last epoch switch. Epoch markers are added to an additional, reserved category named EM. By convention, EM is contained in all excerpts.

4.1 Formal Description

We introduce some additional notation. When signing multiple counter values, we will sign a partial map $f\colon \{0,1\}^* \to \mathbb{N}_0$, which is formally modelled as a *set* f *of pairs* $(\nu, c_\nu) \in \{0,1\}^* \times \mathbb{N}_0$, signifying that the counter value of category ν is c_ν, or $f(\nu) = c_\nu$. For each category name ν, there is at most one pair in f that has ν as the first component. We also write such partial maps as $\{\nu_0 \mapsto c_{\nu_0}, \ldots, \nu_n \mapsto c_{\nu_n}\}$. A *key* of f is a bit string $\nu \in \{0,1\}^*$ for which $f(\nu)$ is defined. The set of keys for f is $\mathrm{keys}(f) := \{\nu \in \{0,1\}^* \mid \exists c \in \mathbb{N}_0 : (\nu, c) \in f\}$.

We assume that SALVE uses an efficient encoding scheme to map pairs to bit strings. We require that there are no pairs (f, m') and (N, E) (where $m' \in \{0,1\}^*$, f is a partial mapping $f\colon \{0,1\}^* \to \mathbb{N}_0$, N is a finite set of bit strings, and E is a sequence of categorized log messages) that are encoded to the same bit string.

SALVE. Let $\Sigma_{\mathrm{FS}} = (\mathrm{KeyGen}_{\mathrm{FS}}, \mathrm{Update}_{\mathrm{FS}}, \mathrm{Sign}_{\mathrm{FS}}, \mathrm{Verify}_{\mathrm{FS}})$ be a key-evolving signature scheme. The key-evolving categorized audit log scheme SALVE is given by the following algorithms:

KeyGen(T)

creates a key pair by running $(sk_0, pk) \leftarrow \mathrm{KeyGen}_{\mathrm{FS}}(T+1)$. The initial signature is the empty sequence $\sigma_{0,-1} := \langle\rangle$. The output is $(sk_0, pk, \sigma_{0,-1})$.

AppendAndSign($sk_i, M_{0,j-1}, m_j = (N_j, m'_j), \sigma_{0,j-1}$)

is called to create a new signature $\sigma_{0,j}$ when a new log entry $m_j = (N_j, m'_j)$ is appended to the current log file $M_{0,j-1} = \langle m_0, \ldots, m_{j-1}\rangle$. Besides $M_{0,j-1}$ and m_j, it also receives the current secret key sk_i and the current signature $\sigma_{0,j-1}$ as input.

We assume EM $\notin N_j$, except when AppendAndSign is called from the Update algorithm (see below), and All $\in N_j$.

AppendAndSign first determines the current counter values c_ν for all $\nu \in N_j$ (the total count of all log entries previously added to these categories). These counter values may be cached or determined by searching for the most recent log entry added to each category. Let $c_\nu := 0$ if the category ν has never occurred before.

Next, AppendAndSign creates the partial map $f_j = \{\nu \mapsto c_\nu \mid \nu \in N_j\}$, computes $\sigma'_j \leftarrow \mathrm{Sign}_{\mathrm{FS}}(sk_i, (f_j, m'_j))$, and appends $\sigma_j := (f_j, \sigma'_j)$ to $\sigma_{0,j-1}$ to obtain $\sigma_{0,j} := \langle \sigma_0, \ldots, \sigma_{j-1}, \sigma_j\rangle$. It outputs $\sigma_{0,j}$.

Update$(sk_i, M_{0,j-1}, \sigma_{0,j-1})$

is called at the end of each epoch i with the current secret key sk_i, the current log file $M_{0,j-1}$ and the current signature $\sigma_{0,j-1}$. It has two tasks: it must append an epoch marker to $M_{0,j-1}$ (and its accompanying signature to $\sigma_{0,j-1}$) and update the secret key.

In order to create the epoch marker, it determines the set N of all categories that have received a new log entry during epoch i and the total number of log entries c_ν in each of these categories (including log entries from previous epochs). Again, this information may be cached. It then creates the set of all these counters $f'_j := \{\nu \mapsto c_\nu \mid \nu \in N\}$ and encodes $(\text{“End of epoch ”} \parallel i, f'_j) =: m'_j$ as a bit string m'_j in some unique fashion. The epoch marker (which is a categorized log entry) is set to $m_j := (\{\text{All}, \text{EM}\}, m'_j)$ and appended to $M_{0,j-1}$. Next, the Update algorithm computes a signature $\sigma_{0,j} \leftarrow \text{AppendAndSign}(sk_i, M_{0,j-1}, m_j, \sigma_{0,j-1})$ for the log file including the epoch marker m_j.

Finally, if $i < T$, Update computes $sk_{i+1} \leftarrow \text{Update}_{\text{FS}}(sk_i)$, securely erases sk_i and outputs sk_{i+1}. Otherwise it deletes sk_i and outputs $sk_{i+1} := \perp$.

Extract$(sk_i, M_{0,j}, \sigma_{0,j}, N)$

is tasked to create a signature for the excerpt $E(N)$ from the log file $M_{0,j}$ and the signature $\sigma_{0,j} = \langle \sigma_0, \ldots, \sigma_j \rangle$. We assume that we always have $\text{EM} \in N$. The signature mostly consists of the individual signatures for all log messages in the excerpt, including the epoch markers, but also contains a newly generated signature for the entire excerpt. More formally, let $K := I(N, M_{0,j}), l := |K|$. Then Extract computes the signature $\sigma_E \leftarrow \text{Sign}_{\text{FS}}(sk_i, (N, E))$, and outputs $\sigma := \langle \sigma_{k_1}, \ldots, \sigma_{k_l}, \sigma_E \rangle$ as the signature for E.

Verify(pk, N, E, σ)

must check the correctness of the excerpt $E = \langle (N_0, m'_0), \ldots (N_{l-1}, m'_{l-1}) \rangle$ (with $l \in \mathbb{N}_0$) for the categories N based on the public key pk and the signature $\sigma = \langle (f_0, \sigma'_0), \ldots, (f_{l-1}, \sigma'_{l-1}), \sigma_E \rangle$. We assume that we always have $\text{EM} \in N$. If $\text{EM} \notin N$, the signature is rejected as invalid.

The algorithm will use counters c'_ν for all categories $\nu \in N$ to keep track of the number of log entries in each that have been contained in the excerpt. These counters will be compared with the actual counters from the signatures. As a first step, Verify initializes its counters $c'_\nu := 0$ for all $\nu \in N$. If $\text{All} \notin N$, it also sets $c'_{\text{All}} := 0$. It then performs the following checks for each entry $m_j \in E$, in the order of increasing j:

– It checks whether the signature for the individual log entry is valid:

$$\text{Verify}_{\text{FS}}(pk, (f_j, m'_j), c'_{\text{EM}}, \sigma'_j) = 1 \quad , \tag{1}$$

– whether m_j belongs to one of the requested categories:

$$N_j \cap N \neq \emptyset \quad , \tag{2}$$

– whether m_j's set of category names N_j is unchanged:

$$\text{keys}(f_j) = N_j \quad \text{, and} \tag{3}$$

– whether the counter values signed together with the message are as expected:

$$f_j(\nu) = c'_\nu \text{ for all } \nu \in N \cap N_j \ . \tag{4}$$

– If All $\notin N$, it checks whether

$$f_j(\text{All}) \geq c'_{\text{All}} \tag{5}$$

and sets $c'_{\text{All}} := f_j(\text{All}) + 1$.

– If m_j is an epoch marker, i.e. EM $\in N_j$, then Verify decodes m'_j to reconstruct f'_j. It then checks whether

$$f'_j(\nu) = c'_\nu \text{ for all } \nu \in \text{keys}(f'_j) \cap N \ . \tag{6}$$

If any of these checks fail, Verify outputs 0. If they pass, Verify increments c'_ν by one for all $\nu \in N \cap N_j$. The verification procedure then continues with the next j, until (including) $j = l - 1$.

– Finally, Verify checks whether

$$\text{Verify}_{\text{FS}}(pk, (N, E), c'_{\text{EM}}, \sigma_E) \stackrel{?}{=} 1 \ , \tag{7}$$

and outputs 1 if so, and 0 otherwise.

A few notes are in order here:

1. Firstly, observe that for all log entries m_j, the number of epoch markers c_{EM} in the log file (or an excerpt) before m_j is identical to the number i of the epoch in which m_j was signed.
2. Excerpts created by SALVE are signed with the most recent secret key available. The verification algorithm implicitly checks for truncation attacks by using the number of epoch markers in the excerpt as the assumed epoch in which the excerpt has been created (see equation 7). Thus, the final signature σ_E serves as an implicit proof that the signer knows the key of epoch c'_{EM}. Truncating a log file (or an excerpt) to an epoch before the break-in therefore requires forging a σ_E supposedly created with a previous secret key, and thus breaking the security of Σ_{FS}.
3. If the verification algorithm had the current epoch number i as an additional trusted input, it could also check whether $i = c'_{\text{EM}}$. This would strengthen the verification algorithm considerably.
4. Generally, given an excerpt E for some set of categories N, it is easy to create an excerpt for a subset of these categories, or to add other categories to E. However, creating a valid signature σ for the new excerpt is hard, because the set of category names N is included in the signature $\sigma_E \leftarrow \text{Sign}_{\text{FS}}(sk_i, (N, E))$. We view this as a feature, as it prevents an attacker from tampering with excerpts.
5. Much information required by the above algorithms (e.g. current counter values and the set of categories modified since the last epoch switch) can be cached by an implementation. This way, our scheme can be implemented very efficiently.

6. If we want SALVE to support T epochs, the underlying forward-secure signature scheme Σ_{FS} must support $T+1$ epochs. SALVE uses the secret keys of the first T epochs of Σ_{FS} to actually sign log entries. When the last of these epochs is over, the log file is closed and can not take any more log entries. The secret key of the remaining epoch supported by Σ_{FS} is then used to sign excerpts from the closed log file.

Example 4 (Signing and Updating). We return to our bank example. When the log file is created, the KeyGen algorithm creates a pair of keys (sk_0, pk) and initializes the signature $\sigma_{0,-1} := \langle\rangle$ for the empty log file $M_{0,-1} = \langle\rangle$.

Let $m_0 := (N_0 = \{\text{All}, \text{“customer id 1”}, \text{“account creation”}\}, m_0')$ be the first entry added to the log file. The new log file is $M_{0,0} = \langle m_0 \rangle$. The AppendAndSign algorithm is called to create a signature for $M_{0,0}$.

It first determines the number of log entries in the categories $\nu \in N_0$ so far. Since there have been no log entries before, we have $c_{\text{All}} = 0$, $c_{\text{customer id 1}} = 0$ and $c_{\text{account creation}} = 0$.

It therefore sets $f_0 := \{\text{All} \mapsto 0, \text{“customer id 1”} \mapsto 0, \text{“account creation”} \mapsto 0\}$, and stores $\sigma_0 := (f_0, \sigma_0' \leftarrow \text{Sign}_{\text{FS}}(sk_0, (f_0, m_0')))$ as the individual signature for the log entry m_0. The signature for $M_{0,0}$ is $\langle \sigma_0 \rangle$.

Now let $m_1 := (N_1 = \{\text{All}, \text{“customer id 1”}, \text{“deposit”}\}, m_1')$ be the second log entry. When this log entry is added to $M_{0,0}$, we get $M_{0,1} = \langle m_0, m_1 \rangle$.

Again, one needs to create a signature for m_1 (and the new log file $M_{0,1}$). In order to compute the signature for m_1, the AppendAndSign algorithm determines the counter values $c_{\text{All}} = 1$, $c_{\text{customer id 1}} = 1$ and $c_{\text{deposit}} = 0$. These are transformed into $f_1 := \{\text{All} \mapsto 1, \text{“customer id 1”} \mapsto 1, \text{“deposit”} \mapsto 0\}$. The signature for m_1 is $\sigma_1 := (f_1, \text{Sign}_{\text{FS}}(sk_0, (f_1, m_1')))$. This is appended to $\sigma_{0,0}$ to obtain $\sigma_{0,1} = \langle \sigma_0, \sigma_1 \rangle$, the signature for $M_{0,1}$.

Now suppose there is an epoch switch from epoch 0 to epoch 1. The Update algorithm is called. It first collects the counter values of all categories that have had a log entry added to them in epoch 0. These counter values are $c_{\text{All}} = 2$, $c_{\text{customer id 1}} = 2$, $c_{\text{account creation}} = 1$, $c_{\text{deposit}} = 1$, and encodes them to $f_2' := \{\text{All} \mapsto 2, \text{“customer id 1”} \mapsto 2, \text{“account creation”} \mapsto 1, \text{“deposit”} \mapsto 1\}$. It then encodes the tuple $(\text{“end of epoch 0”}, f_2')$ as a bit string m_2'. This bit string is converted to a categorized log message $m_2 := (N_2 = \{\text{All}, \text{EM}\}, m_2')$ by assigning it to the categories All and EM.

Next, m_2 is to be appended to the log file. The Update algorithm computes the new signature $\sigma_{0,2}$ as before: It determines the counter values $c_{\text{All}} = 2$, $c_{\text{EM}} = 0$, and sets $f_2 := \{\text{All} \mapsto 2, \text{EM} \mapsto 0\}$. It then creates the signature $\sigma_2' \leftarrow \text{Sign}_{\text{FS}}(sk_0, (f_2, m_2'))$ and appends $\sigma_2 := (f_2, \sigma_2')$ to $\sigma_{0,1}$. The result is $\sigma_{0,2} = \langle \sigma_0, \sigma_1, \sigma_2 \rangle$. Observe that since m_2' contains f_2' and m_2' has been signed, the number of log entries in all categories is authenticated with sk_0.

Before Update terminates, it evolves sk_0 to $sk_1 \leftarrow \text{Update}_{\text{FS}}(sk_0)$, and securely erases sk_0.

Now assume that one adds two messages in epoch 1: The first one is $m_3 := (N_3 = \{\text{All}, \text{“customer id 2”}, \text{“account creation”}\}, m_3')$ and the second is $m_4 := (N_4 = \{\text{All}, \text{“customer id 1”}, \text{“withdrawal”}\}, m_4')$. The corresponding counters are $f_3 = \{\text{All} \mapsto 3, \text{“customer id 2”} \mapsto 0, \text{“account creation”} \mapsto 1\}$ and $f_4 = \{\text{All} \mapsto 4, \text{“customer id 1”} \mapsto 2, \text{“withdrawal”} \mapsto 0\}$. We skip to the next epoch switch, as the signatures σ_3 and σ_4 are created as above.

At the epoch switch from epoch 1 to epoch 2, Update is called. It first constructs

$$f'_5 = \{\text{All} \mapsto 5, \text{“account creation”} \mapsto 2, \text{“customer id 1”} \mapsto 3, \\ \text{“customer id 2”} \mapsto 1, \text{“withdrawal”} \mapsto 1\} \ .$$

Observe that the counter for the category “deposit” is not contained in f'_5, since there was no log entry in that category during epoch 1. Update creates a categorized log message m_5 from f'_5, signs it (resulting in σ_5), and appends m_5 and σ_5 to the log file $M_{0,4}$ and the signature so far $\sigma_{0,4}$, respectively. It then computes $sk_2 \leftarrow \text{Update}(sk_1)$, deletes sk_1 in an unrecoverable fashion and outputs sk_2.

Example 5 (Excerpts and Verification). Say someone requested an excerpt for any log entries regarding customer 2. Then one creates an excerpt for the categories $N = \{\text{“customer id 2”}, \text{EM}\}$. (Recall that by convention, we have $\text{EM} \in N$ when the extraction algorithm is called.)

The excerpt to be output is $E := \langle m_2, m_3, m_5 \rangle$, since $m_2, m_5 \in C(\text{EM})$ and $m_3 \in C(\text{“customer id 2”})$. Thus, the signature σ for E contains σ_2, σ_3 and σ_5. The last component of σ is a signature σ_E for N and the sequence that is E. This last component is necessary to prevent attackers not having a secret key from freely “combining” signatures for different excerpts. For example, without the additional signature over all log entries in E, if an attacker had signatures for excerpts for the categories N_1 and N_2, then it were trivial to create a signature for the adversary to create a signed excerpt for $N_1 \cup N_2$.

The verification algorithm gets $\langle m_2, m_3, m_5 \rangle$ and $\langle \sigma_2, \sigma_3, \sigma_5, \sigma_E \rangle$ as input, along with the excerpt signature σ and the public key pk. It verifies whether σ_2, σ_3 and σ_5 are valid for m_2, m_3, m_5 using $\text{Verify}_{\text{FS}}$. Note that all epoch markers are included in the excerpt, so Verify can determine the epoch in which these messages were signed by counting the number of epoch markers occuring before the respective message. (In our description above, this is just c_{EM}.)

The verification algorithm also checks whether $\text{keys}(f_j) = N_j$. To understand this, observe that N_j is not signed directly during the signature algorithm, but implicitly (since f_j is signed). If one omitted this check, an adversary might tamper with the categories N_j of the excerpt without the verification algorithm detecting this.

Verify also checks that all counters in f_j match the expected values.

As a last step, Verify checks the signature over the entire excerpt E together with the set of categories N for which this excerpt was created. For this check, it determines the epoch number based on the number of epoch markers in the excerpt.

Lemma 1. SALVE *is correct.*

Proof. We need to show that all checks of Verify pass, when Verify is called with a regularly created signature $\sigma = \langle \sigma_0, \ldots, \sigma_l, \sigma_E \rangle$.

First let us gather some simple observations:

1. Verify correctly counts the number of entries it has seen for each category $\nu \in N$ as c'_ν. c'_ν is also the sequence number expected to be found in the next log message belonging to category ν.
2. In particular, c'_{EM} contains the number of epoch markers it has encountered so far, which is equal to the epoch during which the next message should have been signed (see note 1 on page 20).
3. Similar to observation 1, c'_{All} is the minimum sequence number in the category All expected to be found next.

Now let us show that the checks of Verify pass. For each $j \in \{0, \ldots, k\}$, check 1 will pass with overwhelming probability, due to the correctness of Σ_{FS}, and because of observation 2.

Check 2 will always hold true, because Extract only considers messages that are contained in the excerpt, cf. Definition 6. Check 3 will also pass, because of the construction of f_j in the AppendAndSign algorithm.

Check 4 will pass because for each $\nu \in N_j$, AppendAndSign has set $f_j(\nu)$ to the number of log entries contained in category ν, all of these entries are contained in the excerpt, and Verify counts these (as c'_ν) correctly.

A similar argumentation shows that check 6 is successful.

If All $\notin N$, check 5 verifies that the counters for the category All that are signed together with each log entry form a strictly increasing sequence. (If All $\in N$, this is already verified by check 4. Furthermore, check 4 also verifies that the counter values are consecutive.) This is always the case for excerpts created by the regular mechanism, so this check will never fail.

Finally, equation 7 will be fulfilled with overwhelming probability, because of the correctness of Σ_{FS}.

In total, Verify will only reject a signature if one of the calls to $\mathrm{Verify}_{\mathrm{FS}}$ outputs 0. For each $j \in \{0, \ldots, l\}$, let A_j be the event that $\mathrm{Verify}_{\mathrm{FS}}$ outputs 0 in check 1 for j, and let A_E be the event that $\mathrm{Verify}_{\mathrm{FS}}$ outputs 0 in check 7. Applying a union bound, we get

$$\begin{aligned}
\Pr[\text{Verify outputs } 0] &= \Pr[A_0 \vee \ldots \vee A_l \vee A_E] \\
&\leq \Pr[A_E] + \sum_{j=0}^{l} \Pr[A_j].
\end{aligned}$$

Since each of the probabilities $\Pr[A_j]$ and $\Pr[A_E]$ is negligible, and l is bounded by a polynomial in the security parameter, the result is negligible as well. This means that $\Pr[\text{Verify outputs } 1]$ is overwhelming.

In particular, if Σ_{FS} has perfect correctness (i.e. $\mathrm{Verify}_{\mathrm{FS}}$ always accepts a regularly created signature), then $\Pr[\text{Verify outputs } 0] = 0$, and therefore $\Pr[\text{Verify outputs } 1] = 1$. □

4.2 Security Analysis

We now analyse the security of our scheme above. The following theorem states our main result:

Theorem 1 (Security of SALVE). *If there exists a PPT attacker $\mathcal{A}$ that wins the FS-EUF-CLMA experiment against* SALVE *with probability $\varepsilon_{\mathcal{A}}$, then there exists a PPT attacker $\mathcal{B}$ that wins the FS-EUF-CMA game against Σ_{FS} with probability $\varepsilon_{\mathcal{B}} = \varepsilon_{\mathcal{A}}$.*

Proof. Let $\mathcal{A}$ be an attacker having success probability $\varepsilon_{\mathcal{A}}$ in the FS-EUF-CLMA experiment against SALVE. We construct an adversary $\mathcal{B}$ that tries to break the FS-EUF-CMA-security of the underlying scheme Σ_{FS}, using $\mathcal{A}$ as a component.

Therefore, $\mathcal{B}$ must simulate the FS-EUF-CLMA-experiment with SALVE for $\mathcal{A}$. $\mathcal{B}$ does this as follows.

$\mathcal{B}$ receives a public key pk and the number of epochs T as input. It sets $i := 0$, $j := 0$, $M_{0,-1} := \langle\rangle$, $\sigma_{0,-1} := \langle\rangle$. It then starts executing $\mathcal{A}$ with input $(pk, T-1, \sigma_{0,-1})$.

When $\mathcal{A}$ issues an oracle query, $\mathcal{B}$ reacts as follows:

Signature Queries

When $\mathcal{A}$ requests that a new message $m_j = (N_j, m'_j)$ shall be added to the log file, $\mathcal{B}$ collects the counter values c_ν for all $\nu \in N_j$, initializing them to 0 if the category ν has not occured before. It builds $f_j := \{\nu \mapsto c_\nu \mid \nu \in N_j\}$ and submits (f_j, m'_j) to the signature oracle in the FS-EUF-CMA-experiment. This oracle answers with a signature σ'_j for (f_j, m'_j). $\mathcal{B}$ combines this with f_j to get $\sigma_j := (f_j, \sigma'_j)$. Then $\mathcal{B}$ sets $\sigma_{0,j} := \sigma_{0,j-1} \parallel \sigma_j$, $M_{0,j} := M_{0,j-1} \parallel m_j$, returns $\sigma_{0,j}$ to $\mathcal{A}$, and increments $j := j + 1$.

Excerpt queries

When $\mathcal{A}$ requests a signature for an excerpt for the categories N, $\mathcal{B}$ proceeds as follows.

$\mathcal{B}$ first builds $E(N, M_{0,j})$. Next, $\mathcal{B}$ collects the individual signatures σ_k for all $m_k \in E$. (More formally, let $l = |E|$, and let $I(N, M_{0,j}) = \langle k_1, \dots, k_l\rangle$ again denote the index sequence of the excerpt E with respect to $M_{0,j}$.) $\mathcal{B}$ submits (N, E) to the signature oracle in the FS-EUF-CMA experiment to obtain σ_E. It returns $\sigma = \langle \sigma_{k_1}, \dots, \sigma_{k_l}, \sigma_E\rangle$ to $\mathcal{A}$.

Epoch Switching

When $\mathcal{A}$ requests an epoch switch from epoch i to epoch $i+1$ in the FS-EUF-CLMA experiment, $\mathcal{B}$ creates the epoch marker just as in the Update algorithm: It first determines the set N of categories that had a log entry added to them during epoch i, collects the counters c_ν for all $\nu \in N$, builds $f_j := \{\nu \mapsto c_\nu \mid \nu \in N\}$ and sets $m'_j := (\text{“End of epoch ”} \parallel i, f)$. It then simulates the AppendAndSign algorithm for $m_j := (\{\mathrm{All}, \mathrm{EM}\}, m'_j)$ as described above and obtains a signature σ_j for m_j. It updates the log file and the signature to $M_{0,j} := M_{0,j-1} \parallel m_j$ and $\sigma_{0,j} := \sigma_{0,j-1} \parallel \sigma_{0,j}$,

Finally, it calls the epoch switching oracle in the FS-EUF-CMA-experiment, and increments $i := i + 1$. It returns $M_{0,j}$ and $\sigma_{0,j}$ to $\mathcal{A}$.

Breaking In

When $\mathcal{A}$ requests the current secret key sk_i in the FS-EUF-CLMA-experiment, $\mathcal{B}$ obtains it from its own oracle in the FS-EUF-CMA-experiment and passes it to $\mathcal{A}$.

It is easy to see that the joint distribution of all values occuring in $\mathcal{B}$'s simulation of the FS-EUF-CLMA-experiment ($\mathcal{A}$'s "view") matches the distribution in the real FS-EUF-CLMA-experiment.

At the end of the experiment, $\mathcal{A}$ outputs a forged excerpt E^*, a set of categories N^* and a forged signature σ^* for E^*. If $\mathcal{A}$ outputs an invalid or trivial forgery, then $\mathcal{B}$ outputs $\perp$ and aborts. Otherwise, $\mathcal{B}$ determines which of the following cases has occured and acts as described for each case. For this distinction, we let c^*_{EM} be the number of log entries (N^*_j, m'^*_j) in E^* with $\mathrm{EM} \in N^*_j$.

Case 1: E^* contains $c^*_{\mathrm{EM}} < i_{\mathrm{BreakIn}}$ epoch markers.

Note that this case also captures the event that $\mathcal{A}$ does not obtain a secret key at all (because then $i_{\mathrm{BreakIn}} = \infty$).

In this case, $\mathcal{B}$ outputs $m^* := (N^*, E^*)$ as its message, the number $i^* := c^*_{\mathrm{EM}}$ of epoch markers in E^* as the epoch number, and the last element σ^*_E of the sequence σ^* as its forged signature for m^*. σ^*_E must be a valid signature for (N^*, E^*), since otherwise Verify would have rejected the signature σ^* after checking equation 7.

All queries that $\mathcal{B}$ submitted to its signature oracle during epoch c_{EM} (if any) were either of the form (f_j, m'_j) for some messages (including epoch markers) $m_j = (N_j, m'_j)$ or of the form (N, E) for extraction queries. Because of the encoding, all of $\mathcal{B}$'s signature queries (f_j, m'_j) for log messages (N_j, m'_j) differ from (N^*, E^*) (which is a tuple of a set of bitstrings and a sequence of categorized log messages) . Also, since E^* is a non-trivial forgery in the FS-EUF-CLMA game, $\mathcal{B}$ did never request a signature for (N^*, E^*) in epoch i^*. Finally, since $i^* < i_{\mathrm{BreakIn}}$, $\mathcal{B}$'s output is a non-trivial forgery in the FS-EUF-CMA experiment.

Hence, $\mathcal{B}$'s output is valid and non-trivial, so $\mathcal{B}$ wins the FS-EUF-CMA game.

Case 2: E^* contains $c^*_{\mathrm{EM}} \geq i_{\mathrm{BreakIn}}$ epoch markers.

Let M_i and E_i be as in Definition 8, that is, M_i is the log file returned by $\mathcal{A}$'s most recent call to the epoch switching oracle, and E_i is the excerpt for the categories N^* of M_i. Observe that if $\mathcal{A}$ broke in during epoch $i_{\mathrm{BreakIn}} = 0$, then we had $M_i = \langle\rangle$ by definition, and so $E_i = \langle\rangle$, which is a prefix of *all* excerpts E^* that $\mathcal{A}$ may have created. Thus, any forgery of $\mathcal{A}$ were trivial, and $\mathcal{A}$ could not win the game. In the following, we may therefore assume $i_{\mathrm{BreakIn}} > 0$.

Let E^*_i be the prefix of E^* up until (including) the i_{BreakIn}-th epoch marker (the i_{BreakIn}-th log message (N^*_j, m'^*_j) with $\mathrm{EM} \in N^*_j$). We know that E_i is not a prefix of E^*_i, since otherwise E_i would also be a prefix of E^* in contradiction to $\mathcal{A}$'s forgery not being trivial.

Let $E_i = \langle m_j \rangle_{j=0}^{l-1}$, $E^*_i = \langle m^*_j \rangle_{j=0}^{l^*-1}$, $m^*_j = (N^*_j, m'^*_j)$ for all $j \in \{0, \ldots, l^* - 1\}$ and $m_j = (N_j, m'_j)$ for all $j \in \{0, \ldots, l-1\}$. $\mathcal{B}$ builds the sequences $S^* =$

$\langle(f_0^*, m_0'^*), \ldots, (f_{l^*-1}^*, m_{l^*-1}'^*)\rangle$ (taking the f_j^* from the signatures $\sigma_j^* \in \sigma^*$) and $S = \langle(f_0, m_0'), \ldots, (f_{l-1}, m_{l-1}')\rangle$ (taking the f_j from the signatures σ_j it constructed during the simulation). Note that S contains exactly $\mathcal{B}$'s oracle queries during epochs 0 through $i_{\text{BreakIn}} - 1$, restricted to those messages that belong to at least one of the categories N^*. Also observe that $S^* \neq S$, since we otherwise had $E_i^* = E_i$ (by equations 2 and 3), in contradiction to E_i not being a prefix of E_i^*.

The key observation is that there must be a $(f_k^*, m_k'^*) \in S^*$ with $(f_k^*, m_k'^*) \notin S$ ($k \in \{0, \ldots, l^* - 1\}$). Suppose for the sake of a contradiction that there is no such pair. Then S^* consists entirely of pairs that also occur in S. Obviously, S^* can not contain duplicate pairs $(f_k^*, m_k'^*)$, since the verification algorithm would have rejected the excerpt when checking that counters always increase (equations 4 and/or 5). Since S^* contains only pairs also contained in S, contains no duplicates, and $S^* \neq S$, S^* is missing at least one tuple from S. If S^* is missing an epoch marker from S, but contains no duplicates and no new epoch markers, then the number of epoch markers in S^* is at most $i_{\text{BreakIn}} - 1$, in contradiction to the construction of S^* (which contains exactly i_{BreakIn} epoch markers). So S^* is missing some regular log entry. But then Verify had failed when checking the counters in equation 6, which is impossible if $\mathcal{A}$'s output was valid.

So we have established that S^* contains a pair $(f_k^*, m_k'^*) \notin S$. $\mathcal{B}$ searches for this pair, and outputs it as the message. It also outputs the number of epoch markers in S^* before $(f_k^*, m_k'^*)$ as the epoch number i^* and $\sigma_k'^*$ as the signature.

This is a valid signature, since equation 1 holds. It remains to show that this is a non-trivial forgery. Firstly, the number of epoch markers before $(f_k^*, m_k'^*)$ is at most $i_{\text{BreakIn}} - 1$, so the signature $\sigma_k'^*$ is valid for an epoch $i^* < i_{\text{BreakIn}}$. Secondly, $\mathcal{B}$ has never requested $(f_k^*, m_k'^*)$ from its signature oracle, since $(f_k^*, m_k'^*) \notin S$, where S is exactly the set of $\mathcal{B}$'s signature queries for all messages belonging to at least one of the categories N^*, such as m_k^*. Hence, $\mathcal{B}$ wins the FS-EUF-CMA game in case 2, since it outputs a non-trivial and valid forgery.

Since $\mathcal{B}$'s simulation of the FS-EUF-CLMA game for $\mathcal{A}$ is perfect, $\mathcal{B}$ wins both in case 1 and in case 2, and one of these cases occurs whenever $\mathcal{A}$ outputs a valid and non-trivial signature, we have $\varepsilon_{\mathcal{B}} = \varepsilon_{\mathcal{A}}$. Also, $\mathcal{B}$ runs in polynomial time, as $\mathcal{A}$ does. □

Corollary 1. *If Σ_{FS} is FS-EUF-CMA-secure, and* SALVE *uses proper encodings, then* SALVE *is FS-EUF-CLMA-secure.*

4.3 Performance Analysis

In this section, we analyse the runtime and storage overhead of SALVE. Our findings are derived from the algorithms described in section 4.1. Since SALVE can be instantiated with an arbitrary forward-secure signature scheme Σ_{FS}, we give our findings with regard to algorithm runtime in terms of calls to algorithms of Σ_{FS}, and our findings in regard to storage overhead in terms of key and signature sizes of Σ_{FS}, respectively. Table 1 summarizes our findings.

Table 1. Performance characteristics of SALVE in relation to Σ_{FS}. We use sets, sequences and bit strings instead of their size and length, respectively, to relieve notation.

Algorithm	Runtime
KeyGen	$1 \times \mathrm{KeyGen}_{\mathrm{FS}} + \mathcal{O}(1)$
AppendAndSign	$1 \times \mathrm{Sign}_{\mathrm{FS}} + \mathcal{O}(N_j(\log N_j + \log N_{\mathrm{total}}) + m'_j)$
Update	$1 \times \mathrm{Update}_{\mathrm{FS}} + 1 \times \mathrm{Sign}_{\mathrm{FS}} + \mathcal{O}(N_{\mathrm{epoch}} \log N_{\mathrm{total}})$
Extract	$1 \times \mathrm{Sign}_{\mathrm{FS}} + \mathcal{O}(R \log N)$
Verify	$(E+1) \times \mathrm{Verify}_{\mathrm{FS}} + \mathcal{O}(R \log N)$

Datum	Size
Secret Key	$1 \times sk_{\mathrm{FS}} + 0$
Public Key	$1 \times pk_{\mathrm{FS}} + 0$
Log File Signature	$(M+i) \times \sigma_{\mathrm{FS}} + \mathcal{O}(R)$
Excerpt Signature	$(E+i+1) \times \sigma_{\mathrm{FS}} + \mathcal{O}(R)$

Throughout our analysis, let M denote the current log file, i be the current epoch, R be the total number of associations between log entries and categories (i.e. $R := \sum_{j=0}^{|M|-1} |N_j|$), E be the excerpt being signed by the Extract algorithm or verified by Verify, N_{total} be the set of (the names of) all categories that have been used so far, and N_{epoch} be the set of (the names of) the categories that have received a new log entry in the epoch being ended by the update procedure. Our runtime analysis assumes that:

- All sequence numbers c_ν and category names ν have size $\mathcal{O}(1)$, i.e. there is an a-priori-bound on the length of these. We stress that we make this assumption purely to simplify the analysis. Our scheme can handle sequence numbers and category names of arbitrary length.
- The implemenation always stores sets N of category names in an ordered fashion in order to achieve a unique representation. Maps f_j are ordered as well, by N_j.
- The implementation caches sequence numbers in balanced binary trees. In this case, lookup, insertion and update operations to the cache take $\log |N_{\mathrm{total}}|$ time units. This is a conservative assumption, since the same operations have an expected cost of $\mathcal{O}(1)$ time units for hash-table based caches.
- The implementation caches the names of all categories that have received a new log message in the current epoch. Let this set be denoted by N_{epoch}.
- We also assume that encoding and decoding pairs (f_j, σ'_j) to and from $\{0,1\}^*$ takes time $\mathcal{O}(|f_j| + |\sigma'_j|)$.

Algorithm Runtime Analysis

Key Generation.
The runtime of the KeyGen algorithm is dominated by the call to $\mathrm{KeyGen}_{\mathrm{FS}}$, which creates a key for $T+1$ time periods. All other computations can be done in $\mathcal{O}(1)$ time units.

Message Signing.
The AppendAndSign algorithm must determine the current counter values c_ν for all $\nu \in N_j$ in order to create the mapping f_j. We assume that the algorithm first sorts N_j in order to achieve a unique representation. This can be done in $\mathcal{O}(|N_j| \log |N_j|)$ time units. Looking up all counter values takes $\mathcal{O}(|N_j| \log |N_{\mathrm{total}}|)$ time units. Encoding f_j to a binary string takes time $\mathcal{O}(|f_j| + |m'_j|) = \mathcal{O}(|N_j| + |m'_j|)$. The signing of the tuple then takes one call to $\mathrm{Sign}_{\mathrm{FS}}$.

Updating the Secret Key.
The Update algorithm accesses the cached set N_{epoch} and looks up the corresponding counter values c_ν. This takes at most $\mathcal{O}(|N_{\mathrm{epoch}}| \log |N_{\mathrm{total}}|)$ time units. It then calls the AppendAndSign algorithm, and thus inherits its runtime costs. Note that N_j is constant for this call, so $|N_j| = 2$ can be disregarded in the $\mathcal{O}$ notation. Finally, it performs a call to $\mathrm{Update}_{\mathrm{FS}}$.

Extraction of Excerpts.
Extract first sorts N in time $\mathcal{O}(|N| \log |N|)$. It then scans through M to find relevant log entries. For each log entry $m_j = (N_j, m'_j)$, the algorithm can check if $N_j \cap N = \emptyset$ with at most $|N_j|$ lookup operations in N. Thus, scanning the entire log file takes $\mathcal{O}(\sum_{j=0}^{l-1} |N_j| \log |N|) = \mathcal{O}(R \log |N|)$ time units, where $l := |M|$.

Verification.
The verification algorithm takes $|E|+1$ calls to $\mathrm{Verify}_{\mathrm{FS}}$ for checks 1 and 7. Checks 2 and 4 take $\mathcal{O}(|N_j| \log |N|)$ operations per iteration, check 3 only $\mathcal{O}(|N_j|)$. Check 5 can be done in $\mathcal{O}(|f_j|) = \mathcal{O}(|N_j|)$ time units.
For check 6, let $N_{\mathrm{epoch},j}$ be the set of categories that received at least one new entry during epoch j. Then all checks of this type can be implemented in time $\mathcal{O}(\sum_{j=0}^{i-1} |N_{\mathrm{epoch},j}| \log |N|)$.
In total, we have $(|E|+1)$ calls to $\mathrm{Verify}_{\mathrm{FS}}$, and

$$\begin{aligned}
&\mathcal{O}\left(\sum_{j=0}^{l-1} |N_j| \log |N| + \sum_{j=0}^{i-1} |N_{\mathrm{epoch},j}| \log |N|\right) \\
&\quad = \mathcal{O}\left(\left(\sum_{j=0}^{l-1} |N_j| + \sum_{j=0}^{i-1} |N_{\mathrm{epoch},j}|\right) \log |N|\right) \\
&\quad = \mathcal{O}(R \log |N|)
\end{aligned}$$

other operations.

Storage Overhead In the following, we analyze the storage overhead imposed by SALVE.

Key Sizes.

The sizes of SALVE's public and secret keys are the same as Σ_{FS}'s.

Log File Signature Size.

A signature for an log file M consists of $|M| + i$ signatures of Σ_{FS}, as well as the maps f_j, which take $\mathcal{O}(\sum_{j=0}^{l-1} |N_j|) = \mathcal{O}(R)$ bits.

Excerpt Signature Size.

The signature for an excerpt E consists of each log entry's individual signature, including the signatures for all epoch markers, and a final signature on the pair (N, E). We thus have $|E| + i + 1$ signatures of Σ_{FS}. Furthermore, we have $|E| + i$ maps f_j, which take at most $\mathcal{O}(R)$ bits in total.

Comparison to Other Schemes We now compare the efficiency of SALVE to the performance of other schemes in the literature. In particular, we compare to the scheme by Ma and Tsudik [21,22] and the Logcrypt scheme by Holt [16], since both constructions are generically built on an underlying signature scheme, too. We also compare to the BAF [32,33] and LogFAS [34] schemes by Yavuz et al.

However, Ma and Tsudik require a signature scheme that is not only forward-secure, but can also sequentially aggregate signatures, while Holt's scheme uses a *standard* digital signature without special properties such as forward-security or sequential aggregation.[9] SALVE can be seen in between these two, as SALVE requires the underlying signature scheme to be forward-secure, but does not require the aggregation property.

The different requirements on the underlying signature scheme make it very hard to compare these schemes fairly. For example, the aggregate signature scheme used by Ma and Tsudik hides the amount of work required to verify a signature behind just one call to the aggregate verification algorithm. Comparison is complicated further by the issue that both Ma and Tsudik as well as Holt propose to perform an epoch switch every time a log entry has been added. (This is a case in which SALVE performs badly. However, given the linear overheads imposed by LogCrypt and Ma's and Tsudik's schemes, their schemes are not very practical in this case, neither.)

Comparing these three schemes to BAF and LogFAS is even harder, since BAF and LogFAS are not generically built on an arbitrary signature scheme (possibly requiring additional properties), but use very concrete hardness assumptions and constructions. (Actually, LogFAS does use a signature scheme generically, but requires more concrete hardness assumptions in addition.)

Table 2 shows our results. For Logcrypt, SALVE, the scheme by Ma and Tsudik as well as LogFAS, KeyGen, Sign, Verify, Update, Asig and Aver refer to the costs to call the respective underlying signature scheme's algorithm. Similarly, $|sk|, |pk|, |\sigma|$ refer to the sizes of the underlying scheme's secret key, public key and signatures, respectively. For Logcrypt, $n \in \mathbb{N}$ is a parameter that can

[9] Holt implicitly constructs a forward-secure scheme from it by building a long certification chain, that is embedded in the log file. The forward-secure scheme is a simple variant of the "Long Signature" scheme from [5, Section 2].

be chosen freely. For BAF and LogFAS, ModExp, ModMul and ModAdd refer to the costs of modular exponentiation, multiplication and addition respectively, and H refers to the cost of evaluating a hash function on a relatively short input. BigInt refers to the size of a large integer value.[10]

Comparison with Logcrypt and the MT scheme. We see that SALVE is competitive with Logcrypt and the scheme by Ma and Tsudik in terms of key generation time, log entry signing time, as well as secret and public key size. It performs only slightly worse than these schemes for the key evolution and verification algorithms. (All forward-secure sequential aggregate signature schemes that we know of require at least $\mathcal{O}(|M|)$ operations. These operations may be modular squarings or even pairing evaluations.)

In terms of storage overhead for the log file SALVE beats Logcrypt, but can not level with the scheme by Ma and Tsudik, since they use (sequential) *aggregate* signatures.

Note that the aggregation approch by Ma and Tsudik comes with two severe drawbacks: Firstly, their scheme can not verify any log entry individually without verifying the entire log file. Secondly, if a single log entry is modified, verification of the entire log file fails, and *all* information stored in the log file must be considered to be forged by the adversary. Ma and Tsudik recognize these drawbacks, and devise an alternative "immutable" scheme that solves these issues. The modified scheme has $(|M|+1) \times |\sigma|$ storage overhead, which is notably but not far better than SALVE.

Comparison with BAF and LogFAS. As stated before, comparing SALVE to BAF and LogFAS is very hard, since SALVE may have very different performance characteristics depending on the underlying signature scheme Σ_{FS}.

LogFAS is very efficient in log file verification time. We expect SALVE to be slower than LogFAS in this regard. LogFAS also has a very efficient key evolution procedure (because all epoch keys are pre-computed during key generation) and a moderate signature creation time. However, this high efficiency in selected regards is paid for with key generation time and secret key size that are linear in T, and very large signature size. We expect SALVE to easily outperform LogFAS in these parameters.

BAF, in contrast to LogFAS, is heavily optimized for an efficient signing procedure. It also has an efficient key evolution algorithm, a modest secret key size and a very compact signature, that is independent of $|M|$, just as the scheme by Ma and Tsudik. (BAF therefore carries the same backdraws.) These enjoyable performance properties of BAF are paid for with a very expensive key generation algorithm and an extreme public key size.

[10] BAF and LogFAS use prime-order subgroups of a prime field where the discrete logarithm problem is intractable with current methods and equipment. In order not to complicate our analysis further, we do not differentiate between integers in the size of the group order (at least 160 bits) and integers in the size of the prime field size (at least 1024 bits). One may conservatively assume that all of these integers are 160 bits in size, referring only to the group order.

Table 2. Comparison of SALVE with other Secure Logging Schemes.

	Logcrypt	SALVE	Ma and Tsudik	BAF	LogFAS
Algorithm	**Runtime**				
Key Generation	KeyGen	KeyGen	KeyGen	$2T \times$ ModExp $+ 5T \times H$	KeyGen $+ (T+1) \times$ ModExp $+ T \times$ Sign
Log Entry Signing	Sign	Sign	Asig	$2 \times H + 2 \times$ ModAdd	$1 \times H + 1 \times$ ModExp$+$ $2 \times$ (ModMul $+$ ModAdd)
Updating	KeyGen $+ 1/n \times$ Sign[11]	Update $+$ Sign	Update	$2 \times H$	deletion only
Excerpt Signing	—	Sign	—	—	—
Verification	$\|M\| \times$ Verify	$(\|E\| + i + 1) \times$ Verify	Aver	$(\|M\|+1) \times$ ModExp$+$ $(2\,\|M\|-1) \times$ ModMul	$2 \times$ ModExp $+$ $(\|M\|+1) \times$ ModMul
Datum	**Size**				
Secret Key	$\mathcal{O}(n) \times \|sk\|$	$\|sk\|$	$\|sk\|$	$4 \times$ BigInt	$(T-i) \times (5 \times$ BigInt $+ \|\sigma\|)$
Public Key	$\|pk\|$	$\|pk\|$	$\|pk\|$	$(4T+3) \times$ BigInt	$4 \times$ BigInt $+ \|pk\|$
Log File Signature	$(\|M\| + i) \times \|\sigma\| + i \times \|pk\|$	$(\|M\| + i) \times \|\sigma\|$	$\|\sigma\|$	$2 \times$ BigInt	$\|M\| \times (5 \times$ BigInt $+ \|\sigma\|)$
Excerpt Signature	—	$(\|E\| + i + 1) \times \|\sigma\|$	—	—	—

5 Conclusion

It is a desirable feature of secure logging schemes to have verifiable excerpts. We have defined a security notion for such logging schemes, and proposed a new scheme that provably fulfills this notion. Our scheme can be instantiated with an arbitrary forward-secure signature scheme, and can therefore be tuned to specific performance requirements and based on a wide variety of computational assumptions.

Future work will be directed at constructing logging schemes that stop *all* truncation attacks while allowing for verification of individual log entries.

Acknowledgements. I would like to thank Jörn Müller-Quade and my colleagues and friends Alexander Koch, Tobias Nilges and Bernhard Löwe for helpful discussions and remarks. I am also grateful to the anonymous reviewers for their comments. This work was supported by the German Federal Ministry of Education and Research (BMBF) as part of the MisPel program under grant no. 13N12063. The views expressed herein are the author's responsibility and do not necessarily reflect those of BMBF. This is the full version of a paper that appeared at the cryptographer's track of the RSA conference 2016. The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-29485-8_11`.

References

1. Michel Abdalla, Sara Miner, and Chanathip Namprempre. Forward-secure threshold signature schemes. In David Naccache, editor, *Topics in Cryptology — CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 441–456. Springer Berlin Heidelberg, 2001.
2. Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 116–129. Springer Berlin Heidelberg, 2000.
3. Rafael Accorsi. Safe-keeping digital evidence with secure logging protocols: State of the art and challenges. In *Fifth International Conference on IT Security Incident Management and IT Forensics, 2009. IMF '09*, pages 94–110, Sept 2009.
4. N.J. Al Fardan and K.G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540, May 2013.
5. Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer Berlin Heidelberg, 1999.
6. Mihir Bellare and Bennet Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2003.
7. Mihir Bellare and Bennet S. Yee. Forward integrity for secure audit logs. Technical report, University of California at San Diego, 1997.

[11] The values shown here are an average per log entry.

8. Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 416–432. Springer Berlin Heidelberg, 2003.
9. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer Berlin Heidelberg, 2001.
10. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.
11. Xavier Boyen, Hovav Shacham, Emily Shen, and Brent Waters. Forward-secure signatures with untrusted update. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 191–200, New York, NY, USA, 2006. ACM.
12. Common Criteria for Information Technology Security Evaluation, version 3.1 r4, part 2, September 2012. `https://www.commoncriteriaportal.org/cc/`.
13. Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 317–334, Berkeley, CA, USA, 2009. USENIX Association.
14. Ariel Futoransky and Emiliano Kargieman. VCR y PEO: Dos protocolos criptográficos simples. In *25 Jornadas Argentinas de Informática e Investigación Operativa*, 1995. `http://www.coresecurity.com/files/attachments/2Protocolos.pdf`.
15. Ariel Futoransky and Emiliano Kargieman. VCR and PEO revised, 1998. `http://www.coresecurity.com/files/attachments/PEO.pdf`, Accessed on Feb. 18th 2015.
16. Jason E. Holt. Logcrypt: Forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian Workshops on Grid Computing and e-Research – Volume 54*, ACSW Frontiers '06, pages 203–211, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
17. Fei Hu, Chwan-Hwa Wu, and J. D. Irwin. A new forward secure signature scheme using bilinear maps. Cryptology ePrint Archive, Report 2003/188, 2003. `http://eprint.iacr.org/`.
18. Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354. Springer Berlin Heidelberg, 2001.
19. Donald C. Latham, editor. *Department of Defense Trusted Computer System Evaluation Criteria*. US Department of Defense, December 1985. `http://csrc.nist.gov/publications/history/dod85.pdf`.
20. Di Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 341–352, New York, NY, USA, 2008. ACM.
21. Di Ma and Gene Tsudik. A new approach to secure logging. In Vijay Atluri, editor, *Data and Applications Security XXII*, volume 5094 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin Heidelberg, 2008.
22. Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):2:1–2:21, March 2009.
23. Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In Lars R. Knudsen, editor,

Advances in Cryptology — EUROCRYPT 2002, volume 2332 of *Lecture Notes in Computer Science*, pages 400–417. Springer Berlin Heidelberg, 2002.
24. Giorgia Azzurra Marson and Bertram Poettering. Practical secure logging: Seekable sequential key generators. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 111–128. Springer Berlin Heidelberg, 2013.
25. An Introduction to Computer Security: The NIST handbook, October 1995. NIST Special Publication 800-12.
26. Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *The Seventh USENIX Security Symposium Proceedings*, 1998.
27. Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, May 1999.
28. Dawn Xiaodong Song. Practical forward secure group signature schemes. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, pages 225–234, New York, NY, USA, 2001. ACM.
29. Vassilios Stathopoulos, Panayiotis Kotzanikolaou, and Emmanouil Magkos. A framework for secure and verifiable logging in public communication networks. In Javier Lopez, editor, *Critical Information Infrastructures Security*, volume 4347 of *Lecture Notes in Computer Science*, pages 273–284. Springer Berlin Heidelberg, 2006.
30. Brent R. Waters, Dirk Balfanz, Glenn Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *In The 11th Annual Network and Distributed System Security Symposium*, 2004.
31. Wensheng Xu, David W Chadwick, and Sassa Otenko. A pki based secure audit web server. *IASTED Communications, Network and Information and CNIS*, 2005.
32. Attila A. Yavuz and Ning Peng. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 219–228, Dec 2009.
33. Attila A. Yavuz, Ning Peng, and Michael K. Reiter. BAF and FI-BAF: Efficient and publicly verifiable cryptographic schemes for secure logging in resource-constrained systems. *ACM Trans. Inf. Syst. Secur.*, 15(2):9:1–9:28, July 2012.
34. Attila A. Yavuz, Ning Peng, and Michael K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin Heidelberg, 2012.
35. Attila A. Yavuz and Michael K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. Technical Report TR-2011-21, North Carolina State University. Department of Computer Science, September 2011. `http://www.lib.ncsu.edu/resolver/1840.4/4284`.
36. Jianhong Zhang, Qianhong Wu, and Yumin Wang. A novel efficient group signature scheme with forward security. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *Information and Communications Security*, volume 2836 of *Lecture Notes in Computer Science*, pages 292–300. Springer Berlin Heidelberg, 2003.