Efficient, Compromise Resilient and Append-only Cryptographic Schemes for Secure Audit Logging

Attila A. Yavuz, Peng Ning
Department of Computer Science,
North Carolina State University,
Raleigh, NC, USA
{aayavuz,pning}@ncsu.edu

Michael K. Reiter
Department of Computer Science,
University of North Carolina,
Chapel-Hill, NC, USA
reiter@cs.unc.edu

September 2011

Abstract

Due to the forensic value of audit logs, it is vital to provide *compromise resiliency* and *append-only* properties in a logging system to prevent active attackers. Unfortunately, existing symmetric secure logging schemes are not publicly verifiable and cannot address applications that require public auditing (e.g., public financial auditing), besides being vulnerable to certain attacks and dependent on continuous trusted server support. Moreover, Public Key Cryptography (PKC)-based secure logging schemes require Expensive Operations (ExpOps) that are costly for both loggers and verifiers, and thus are impractical for computation-intensive environments.

In this paper, we propose a new class of secure audit logging scheme called *Log Forward-secure and Append-only Signature (LogFAS)*. LogFAS achieves the most desirable properties of both symmetric and PKC-based schemes simultaneously. LogFAS can produce publicly verifiable forward-secure and append-only signatures without requiring any online trusted server support or time factor. Most notably, LogFAS is the only PKC-based scheme that achieves the optimal verifier computational and storage efficiency. That is, LogFAS can verify L log entries with always small and constant number of ExpOps regardless of the value of L. Moreover, each verifier stores only a small and constant-size public key regardless of the number of log entries to be verified or the number of loggers in the system. In addition, a LogFAS variation allows fine-grained verification of any subset of log entries and fast detection of corrupted log entries. All these properties make LogFAS an ideal scheme for secure audit logging in computation-intensive applications.

Keywords: Secure audit logging; applied cryptography; digital forensics; digital signatures; forward security.

1 Introduction

Audit logs have been used to track important events such as user activities and program execution in modern computer systems, providing invaluable information about the state of the systems (e.g., intrusions, crashes). Due to their forensic value, audit logs are an attractive target for attackers. Indeed, an experienced attacker may erase the traces of her malicious activities from the logs, or modify the log entries to implicate other users after compromising the system. Therefore, ensuring the integrity, authenticity and accountability of audit logs in the presence of attackers is critical for any modern computer system [12, 28, 36, 44].

There are straightforward techniques to protect audit logs from active adversaries: (i) Using a tamper resistant hardware on each logging machine to prevent the adversary from modifying audit logs and (ii) transmitting each log entry as soon as it is generated to a remote trusted server. Unfortunately, these approaches have significant limitations as identified in [7, 12, 27, 28, 32, 35, 44]: First, it is impractical to assume both the presence and the "bug-freeness" of a tamper resistant hardware on all types of platforms (e.g., wireless sensors [26], commercial off-the-shelf systems [7]) [12, 25, 27, 28]. Second, it is difficult to guarantee timely communication between each logging machine and the remote trusted server in the presence of active adversaries [15, 27, 44].

The storage industry introduced fast Write-Once-Read-Many (WORM) devices [10, 14], which were soon adopted to protect audit logs [41]. Unfortunately, WORM devices have been shown to be vulnerable to faulty behaviors or insider attacks due to their reliance on software and primitive hardware-based on/off switches [17,32]. This further confirms the danger in simply relying on tamper-resistant hardware.

Cryptographic Log Protection and Desirable Properties: One way to address the above limitations is to develop cryptographic mechanisms that can protect the integrity of audit logs on a physically unprotected machine without relying on tamper-resistant hardware or omni-available and trusted remote servers. There has been extensive research invested in this direction (e.g., [7, 12, 13, 16, 25, 27, 28, 35, 36, 44]). In general, it is desirable for such schemes to have the following properties:

Forward Security and Append-only Properties: Since the log verifiers are not necessarily available to verify the log entries once they are generated, a logger may have to accumulate log entries for a period of time. If the adversary takes full control of the logging machine in this duration, no cryptographic mechanism can prevent her from modifying the post-attack log entries. However, the integrity of log entries accumulated before the attack should be protected (i.e., forward-security property) [1,7,12,16,25,27,28,44]. Furthermore, this protection should not only guarantee the integrity of individual log entries but also the integrity of the log stream as a whole. That is, no selective deletion or truncation of log entries should be possible (i.e., append-only (aggregate) property) [25, 26, 28]).

Note that a plain forward-secure signature (e.g., [1,20,22]) does not provide the append-only property [27,28]. Therefore, a log protection mechanism that solely relies on a plain forward-secure signature (without a signature aggregation strategy) is prone to the tail-truncation attacks [25, 26, 44, 45] (the security model and analysis of the truncation attacks are given in Section 3 and Section 5, respectively). In addition, even the most efficient generic forward-secure constructions (i.e., [29]) require several expensive operations (ExpOps)[1] in signature generation and verification. A straightforward integration of forward-secure signatures (e.g., [23]) with aggregate signatures (e.g., [9]) (to achieve append-only properties as in FssAgg schemes [25]) are also highly computationally costly as discussed below. Hence, the direct adaptation of forward-secure signatures is not sufficient to achieve our goals.

Efficiency: Computational overhead introduced by the secure audit logging scheme must be low at the logger and the log verifier sides.

Public Verifiability: Unlike schemes that enable only a few privileged entities who share secrets with loggers to verify the integrity of log entries, a scheme supporting public verifiability permits any entity to do so. Public verifiability is a desirable property for some critical applications, such as electronic voting, where logs need to be audited and verified by the public [8], and financial applications, where financial books of publicly held companies need to be verified by the current and potential future share holders [16,28].

Provable Security: It is necessary for a scheme to have formal security assessments for its security properties, including attacks specific to audit logging systems such as truncation attacks and delayed detection (details are given in Section 5).

Independence of Online Trusted Server: Ideally such a scheme should not rely on any online trusted server, though an offline trusted server may be used. This is desirable since any dependence on the timely communication with a trusted server could be exploited by an active adversary to defeat the scheme (e.g., delayed detection [27, 28, 44]).

Previous Cryptographic Solutions and Their Limitations: Pioneering forward-secure audit logging schemes [6, 7, 35, 36] rely on symmetric primitives such as Message Authentication Code (MAC) to achieve computationally efficient integrity protection. In these schemes, a trusted server shares a pairwise secret key with each logger before the deployment. Each logger generates a forward-secure MAC of each log entry after the deployment based on this initial key. A small number of privileged semi-trusted entities, who share a partial secret with loggers, can verify the MACs with the aid of the trusted server (only the trusted server has the complete keying information required for the verification).

Despite the simplicity and computational efficiency, the above schemes have significant limitations: First, the symmetric nature of these schemes does not allow public verifiability, and therefore they cannot address applica-

[1] For brevity, in this paper, we refer to an expensive cryptographic operation such as modular exponentiation [40] and pairing [21] as an ExpOp.

Table 1: Comparison of LogFAS schemes and their counterparts for performance, applicability, availability and security parameters

Criteria		*PKC-based*						*SYM* [7,36]
		LogFAS	FssAgg/iFssAgg			BAF	Logcrypt	
Computational			AR	BM	BLS			
On-line	*Sig&Upd (per item)*	$ExpOp$	$ExpOp$			H	$ExpOp$	H
	Ver, (L items)	$ExpOp + O(L \cdot H)$	$O(L \cdot (ExpOp + H))$					$O(L \cdot H)$
	Subset ver (l')	$ExpOp + O(l' \cdot H)$	$O(2l'(ExpOp + H))$			Not immutable		$O(l' \cdot H)$
	Efficient Search	Available	Not Available					-
Key Generation (Offline)		$O(L \cdot ExpOp)$						$O(L \cdot H)$
Storage	*Verifier*	$O(1)\|K\|$	$O(S \cdot \|K\|)$	$O(L \cdot S)\|K\|$				$O(S \cdot \|K\|)$
	Signer	$O(L \cdot (\|D\| + \|K\|))$	$O(L \cdot \|D\|) + O(1)\|K\|$				$O(L \cdot \|K\|)$	$O(L \cdot \|K\|)$
Communication		$O(L \cdot \|D\|)$						
Public Verifiability		Y	Y					N
Offline Server		Y	Y					N
Immediate Verification		Y	Y					N
Immediate Detection		Y	Y					N
Truncation Resilience		Y	Y				N	N

LogFAS is the only PKC-based secure audit logging scheme that can verify $O(L)$ items with $O(1)$ ExpOp; all other similar schemes require $O(L)$ExpOp. Similarly, LogFAS is the only one achieving $O(1)$ key storage on the verifier side, while all other schemes incur either linear or quadratic storage overhead ($S, |D|, |K|$ denote number of signers in the system, the approximate bit lengths of a log entry and unit keying material, respectively). At the same time, LogFAS is the only scheme that enables truncation-free subset verification and sub-linear search simultaneously.

tions requiring public auditing [16,25,26,44]. Second, they require online remote trusted server support, which entails costly maintenance and attracts potential attacks besides being a potential single-point of failures. Finally, these schemes are shown to be vulnerable against the truncation and delayed detection attacks [27,28].

To mitigate the above problems, several PKC-based secure audit logging schemes have been proposed. Logcrypt [16] extends the forward-secure MAC strategy to the PKC-domain, and thus is publicly verifiable without any online trusted server. However, Logcrypt incurs high storage overhead and is still vulnerable to the truncation attack. FssAgg schemes [25,26,28] were later proposed to achieve truncation-free logging with forward-secure and aggregate signatures (where truncation prevention is achieved via the append-only property). Most recently, BAF [44] was developed to achieve computationally efficient log signing and truncation-attack-resistant logging at the same time. Unfortunately, all these schemes suffer from an efficiency problem.

First, all these schemes are costly for loggers (except for BAF [44]) and extremely costly for the log verifiers. BAF is able to provide efficient signer operations; however, verifier operations are still highly expensive in terms of computational and storage overheads. These overheads limit the use of the above schemes. Second, to verify a particular log entry, all these schemes [25–27,44] force log verifiers to verify the entire set of log entries, which entails a linear ExpOps, and failure of this verification does not give any information about which log entry(ies) is (are) responsible for the failure. The iFssAgg schemes [28] mitigate these problems by allowing more fine-grained verification, but they double the signing/verification costs of their base FssAgg schemes to prevent the truncation attack, and the verification of a subset of given log entries still requires linear ExpOps in terms of the size of the given subset. Furthermore, even if a small fraction of log entries are damaged, detecting damaged entry(ies) requires a linear number of ExpOps.

Our Contribution: In this paper, we propose a new secure audit logging scheme, which we call *Log Forward-secure and Append-only Signature (LogFAS)*. We first develop a main LogFAS scheme, and then extend it to provide additional capabilities. The desirable properties of LogFAS are outlined below. The first three properties show the efficiency of LogFAS compared with their PKC-based counterparts, while the other three properties demonstrate the applicability, availability and security advantages over their symmetric counterparts. Table 1 summarizes the above properties and compares LogFAS with its counterparts.

1. *Efficient Log Verification with $O(1)$ ExpOp*: All existing PKC-based secure audit logging schemes (e.g., [16, 25–28,44,45]) require $O(L)$ ExpOps to verify L log entries, which make them costly. LogFAS is the first PKC-

based secure audit logging scheme that achieves signature verification with only a small number of ExpOps (and $O(L)$ hash operations (H). Specifically, LogFAS can verify L log entries with only $O(1)$ ExpOps regardless of the value of L. Therefore, it is much more efficient than all its PKC-based counterparts, and is also comparably efficient with symmetric schemes (e.g., [6, 7, 26, 35, 36]) at the verifier side.

2. *Efficient Fine-grained Verification and Change Detection*: LogFAS allows fine-grained verification with advantages over iFssAgg, the only previous solution for fine-grained verification:

   i. Unlike iFssAgg schemes [28], LogFAS prevents the truncation attack[2] in the presence of individual signatures without doubling the verification cost.

   ii. LogFAS can verify any selected subset with $l' < L$ log entries with $O(1)$ ExpOps, while iFssAgg schemes require $O(2l')$ExpOps.

   iii. LogFAS can identify the corrupted log entries with a *sub-linear* number of ExpOps when most log entries are intact. In contrast, iFssAgg schemes always require a linear number of ExpOps.

3. *Verifier Storage Efficiency with $O(1)$ Overhead*: Each verifier in LogFAS only stores one public key independent of the number of loggers or the number of log entries to be verified. Therefore, it is the most verifier-storage-efficient scheme among all existing PKC-based alternatives. This enables verifiers to handle a large number of log entries and/or loggers simultaneously without facing any storage problem.

4. *Public Verification*: Unlike the symmetric schemes (e.g., [7, 26, 35, 36]), LogFAS can produce publicly verifiable signatures, and therefore it can protect applications requiring public auditing (e.g., e-voting, financial books) [16, 28].

5. *Independence of Online Trusted Server*: LogFAS schemes do not require online trusted server support to enable log verification. Therefore, LogFAS schemes achieve high availability, and are more reliable than the previous schemes that require such support (e.g., [7, 35, 36, 45]).

6. *High Security*: We prove LogFAS to be forward-secure existentially unforgeable against adaptive chosen-message attacks in Random Oracle Model (ROM) [4]. Furthermore, unlike some previous symmetric schemes [7, 35, 36], LogFAS schemes are also secure against both truncation and delayed detection attacks.

The remainder of this paper is organized as follows. Section 2 provides preliminary background information. Section 3 describes the syntax and models used in this paper. Section 4 presents the proposed LogFAS schemes. Sections 5 and 6 provide the security and the performance analysis of the LogFAS schemes, respectively. Section 7 discusses the related work, and Section 8 concludes this paper.

2 Preliminaries

Notation. $||$ denotes the concatenation operation. $|x|$ denotes the bit length of variable x. $x \overset{\$}{\leftarrow} \mathcal{S}$ denotes that variable x is randomly and uniformly selected from set $\mathcal{S}$. For any integer l, $(x_0, \ldots, x_l) \overset{\$}{\leftarrow} \mathcal{S}$ means $(x_0 \overset{\$}{\leftarrow} \mathcal{S}, \ldots, x_l \overset{\$}{\leftarrow} \mathcal{S})$. We denote by $\{0,1\}^*$ the set of binary strings of any finite length. H is an ideal cryptographic hash function, which is defined as $H : \{0,1\}^* \rightarrow \{0,1\}^{|H|}$; $|H|$ denotes the output bit length of H. $\mathcal{A}^{\mathcal{O}_0,\ldots,\mathcal{O}_i}(\cdot)$ denotes algorithm $\mathcal{A}$ is provided with oracles $\mathcal{O}_0, \ldots, \mathcal{O}_i$. For example, $\mathcal{A}^{Scheme.Sig_{sk}}(\cdot)$ denotes that algorithm $\mathcal{A}$ is provided with a *signing oracle* of signature scheme $Scheme$ under private key sk.

Definition 1 *A signature scheme SGN is a tuple of three algorithms* (Kg, Sig, Ver) *defined as follows:*

[2] The truncation attack is a special type of deletion attack, in which the adversary deletes a continuous subset of tail-end log entries. This attack can be prevented via "all-or-nothing" property [26]: The adversary either should remain previously accumulated data intact, or should not use them at all (she cannot selectively delete/modify any subset of this data [28]). LogFAS is proven to be secure against the truncation attack in Section 5.

- $(sk, PK) \leftarrow SGN.Kg(1^\kappa)$*: Key generation algorithm takes the security parameter* 1^κ *as the input. It returns a private/public key pair* (sk, PK) *as the output.*

- $\sigma \leftarrow SGN.Sig(sk, D)$*: The signature generation algorithm takes* sk *and a data item* D *as the input. It returns a signature* σ *as the output (also denoted as* $\sigma \leftarrow SGN.Sig_{sk}(D)$*).*

- $c \leftarrow SGN.Ver(PK, D, \sigma)$*: The signature verification algorithm takes* PK*,* D *and* σ *as the input. It outputs a bit* c*, with* $c = 1$ *meaning* valid *and* $c = 0$ *meaning* invalid.

Definition 2 *Existential Unforgeability under Chosen Message Attack (EU-CMA) experiment for* SGN *is as follows:*
Experiment $Expt_{SGN}^{EU\text{-}CMA}(\mathcal{A})$
$(sk, PK) \leftarrow SGN.Kg(1^\kappa)$, $(D^*, \sigma^*) \leftarrow \mathcal{A}^{SGN.Sig_{sk}(\cdot)}(PK)$,
If $SGN.Ver(PK, D^*, \sigma^*) = 1$ *and* D^* *was not queried, return* 1*, else, return* 0.

EU-CMA-advantage *of* $\mathcal{A}$ *is* $Adv_{SGN}^{EU\text{-}CMA}(\mathcal{A}) = Pr[Expt_{SGN}^{EU\text{-}CMA}(\mathcal{A}) = 1]$.
EU-CMA-advantage *of* SGN*is* $Adv_{SGN}^{EU\text{-}CMA}(t, L, \mu) = \max_{\mathcal{A}}\{Adv_{SGN}^{EU\text{-}CMA}(\mathcal{A})\}$*, where the maximum is over all* $\mathcal{A}$ *having time complexity* t*, making at most* L *oracle queries, and the sum of lengths of these queries being at most* μ *bits.*

LogFAS is built on the Schnorr signature scheme [37]. It also uses an Incremental Hash function $\mathcal{IH}$ [3] and a generic signature scheme SGN (e.g., Schnorr) as building blocks. Both Schnorr and $\mathcal{IH}$ require that $H : \{0,1\}^* \rightarrow \mathbb{Z}_q^*$ is a random oracle.

Definition 3 *The Schnorr signature scheme is a tuple of three algorithms* (Kg, Sig, Ver) *behaving as follows:*

- $(y, \langle p, q, \alpha, Y\rangle) \leftarrow Schnorr.Kg(1^\kappa)$*: Key generation algorithm takes* 1^κ *as the input. It generates large primes* q *and* $p > q$ *such that* $q|(p-1)$*, and then generates a generator* α *of the subgroup* G *of order* q *in* $\mathbb{Z}_p^*$*. It also generates (*$y \overset{\$}{\leftarrow} \mathbb{Z}_q^*$, $Y \leftarrow \alpha^y \bmod p$*), and returns private/public keys* $(y, \langle p, q, \alpha, Y\rangle)$ *as the output.*

- $(s, R, e) \leftarrow Schnorr.Sig(y, D)$*: Signature generation algorithm takes private key* y *and a data item* D *as the input. It returns a signature triplet* (s, R, e) *as follows:*

  $R{\leftarrow}\alpha^r \bmod p,\ e{\leftarrow}H(D||R),\ s{\leftarrow}(r - e \cdot y) \bmod q$*, where* $r \overset{\$}{\leftarrow} \mathbb{Z}_q^*$.

- $c \leftarrow Schnorr.Ver(\langle p, q, \alpha, Y\rangle, D, \langle s, R, e\rangle)$*: Signature verification algorithm takes public key* $\langle p, q, \alpha, Y\rangle$*, data item* D *and signature* $\langle s, R, e\rangle$ *as the input. It returns a bit* c*, with* $c = 1$ *meaning* valid *if* $R = Y^e\alpha^s \bmod p$*, and with* $c = 0$ *otherwise.*

Definition 4 *Given a large random integer* q *and integer* L*, incremental hash function family* $\mathcal{IH}$ *is defined as follows: Given a random key* $z = (z_0, \ldots, z_{L-1})$*, where* $(z_0, \ldots, z_{L-1}) \overset{\$}{\leftarrow} \mathbb{Z}_q^*$ *and hash function* H*, the associated incremental hash function* $\mathcal{IH}_z^{q,L}$ *takes an arbitrary data item set* $D_0, \ldots, D_{L-1}$ *as the input. It returns an integer* $T \in Z_q$ *as the output,*

Algorithm $\mathcal{IH}_z^{q,L}(D_0, \ldots, D_{L-1})$
$T \leftarrow \sum_{j=0}^{L-1} H(D_j)z_j \bmod q$*, return* T.

Target Collision Resistance (TCR) [5] of $\mathcal{IH}$ relies on the intractability of *Weighted Sum of Subset (WSS) problem* [3, 18] assuming that H is a random oracle.

Definition 5 *Given* $\mathcal{IH}_z^{q,L}$*, let* $\mathcal{A}_0$ *be an algorithm that returns a set of target messages, and* $\mathcal{A}_1$ *be an algorithm that returns a bit. Consider the following experiment:*
Experiment $Expt_{\mathcal{IH}_z^{q,L}}^{TCR}(\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1))$
$(D_0, \ldots, D_{L-1}) \leftarrow \mathcal{A}_0(L)$, $z = (z_0, \ldots, z_{L-1}) \overset{\$}{\leftarrow} \mathbb{Z}_q^*$,

$T \leftarrow \mathcal{IH}_z^{q,L}(D_0,\ldots,D_{L-1}), (D_0^*,\ldots,D_{L-1}^*) \leftarrow \mathcal{A}_1(D_0,\ldots,D_{L-1},T,\mathcal{IH}_z^{q,L})$
If $T = \mathcal{IH}_z^{q,L}(D_0^*,\ldots,D_{L-1}^*) \wedge \exists j \in \{0,\ldots,L-1\} : D_j^* \neq D_j$*, return* 1*, else, return* 0.

TCR-advantage *of* $\mathcal{A}$ *is* $Adv_{\mathcal{IH}_z^{q,L}}^{TCR}(\mathcal{A}) = Pr[Expt_{\mathcal{IH}_z^{q,L}}^{TCR}(\mathcal{A}) = 1]$.
TCR-advantage *of* $\mathcal{IH}_z^{q,L}$ *is* $Adv_{\mathcal{IH}_z^{q,L}}^{TCR}(t) = \max_{\mathcal{A}}\{Adv_{\mathcal{IH}_z^{q,L}}^{TCR}(\mathcal{A})\}$*, where the maximum is over all* $\mathcal{A}$ *having time complexity* t*.*

3 Syntax and Models

LogFAS is a Forward-secure and Append-only Signature (FSA) scheme, which combines *key-evolve* (e.g., [2,23]) and *signature aggregation* (e.g., [9, 30]) techniques. Specifically, LogFAS is built on the Schnorr signature scheme [34, 37], and it integrates forward-security and signature aggregation strategies in a novel and efficient way. That is, different from previous approaches (e.g., [25–28, 36, 44, 45]), LogFAS introduces verification with a constant number of ExpOps, selective subset verification and sub-linear search properties via incremental hashing [3] and masked tokens (inspired from [31]) in addition to the above strategies.

Before giving more details, we briefly discuss the *append-only* signatures. A forward-secure and aggregate signature scheme is an *append-only* signature scheme if no message can be re-ordered or selectively deleted from a given stream of messages, while new messages can be appended to the stream [26, 28]. In Section 5, we prove that LogFAS is an append-only signature scheme. Note that, for the envisioned applications, the goal of signature aggregation is to achieve verification with a constant number of ExpOps and append-only property rather than the signer storage efficiency (since the storage overhead is already dominated by the accumulated log entries at the signer side).

Definition 6 *A FSA is comprised of a tuple of three algorithms* $(Kg, FASig, FAVer)$ *behaving as follows:*

- $(sk, PK) \leftarrow FSA.Kg(1^\kappa, L)$*: The key generation algorithm takes the security parameter* 1^κ *and the maximum number of key updates* L *as the input. It returns a private/public key pair* (sk, PK) *as the output.*
- $(sk_{j+1}, \sigma_{0,j}) \leftarrow FSA.FASig(sk_j, D_j, \sigma_{0,j-1})$*: The forward-secure and append-only signing algorithm takes the current private key* sk_j*, a new message* D_j *to be signed and the append-only signature* $\sigma_{0,j-1}$ *on the previously signed messages* $(D_0, \ldots, D_{j-1})$ *as the input. It computes an append-only signature* $\sigma_{0,j}$ *on* $(D_0, \ldots, D_j)$*, evolves (updates)* sk_j *to* sk_{j+1}*, and returns* $(sk_{j+1}, \sigma_{0,j})$ *as the output.*
- $c \leftarrow FSA.FAVer(PK, \langle D_0, \ldots, D_j\rangle, \sigma_{0,j})$*: The forward-secure and append-only verification algorithm takes* PK*,* $\langle D_0, \ldots, D_j\rangle$ *and their corresponding* $\sigma_{0,j}$ *as the input. It returns a bit* c*, with* $c = 1$ *meaning* valid*, and* $c = 0$ *otherwise.*

In LogFAS, private key sk is a vector, whose elements are comprised of specially constructed Schnorr private keys and a set of tokens. These tokens later become the part of append-only signature σ accordingly. The public key PK is a system-wide public key that is shared by all verifiers, and is comprised of two long-term public keys. Details are given in Section 4.

3.1 System Model

LogFAS system model is comprised of a *Key Generation Center (KGC)* and multiple signers (i.e., logging machines that could be compromised) and verifiers. As in forward-secure stream integrity model (e.g., [7, 25, 26]), signers honestly execute the scheme until they are compromised by the adversary. Verifiers may be *untrusted*.

The KGC executes $LogFAS.Kg$ once offline before the deployment, and distributes a distinct private key/token set (auxiliary signature) to each signer, and two long-term public keys to all verifiers. After the deployment, a signer computes the forward-secure and append-only signature of log entries with $LogFAS.FASig$, and verifiers can verify the signature of any signer with $LogFAS.FAVer$ via two public keys without communicating with KGC (constant storage overhead at the verifier side).

In LogFAS, the same logger computes the append-only signature of her own log entries. Note that this form of signature computation is ideal for the envisioned secure audit logging applications, since each logger is only responsible for her own log entries [27,28,44,45]. Signature schemes where the signatures of different signers are aggregated (e.g., [9,30]) is out of the scope of this paper.

3.2 Security Model

A FSA scheme is proven to be *ForWard-secure Existentially Unforgeable against Chosen Message Attack (FWEU-CMA)* based on the experiment defined in Definition 7. In this experiment, $\mathcal{A}$ is provided with two types of oracles that she can query up to L messages in total as follows:

$\mathcal{A}$ is first provided with a *batch signing oracle* $FASig_{sk}(\cdot)$. For each batch query j, $\mathcal{A}$ queries $FASig_{sk}(\cdot)$ on a set of message $\overrightarrow{D}_j$ of her choice once. $FASig_{sk}(\cdot)$ returns a forward-secure and append-only signature $\sigma_{0,j}$ under sk by aggregating σ_j (i.e., the current append-only signature) on $\overrightarrow{D}_j$ with the previous signature $\sigma_{0,j-1}$ on $\overrightarrow{D}_0, \ldots, \overrightarrow{D}_{j-1}$ that $\mathcal{A}$ queried. Assume that $\mathcal{A}$ makes i batch queries (with $0 \leq l \leq L$ individual messages) as described the above until she decides to "break-in".

$\mathcal{A}$ then queries the $Break\text{-}in$ oracle, which returns the remaining $L-l$ private keys to $\mathcal{A}$ (if $l = L$ $Break\text{-}in$ rejects the query).

Definition 7 FWEU-CMA experiment *is defined as follows:*
Experiment $Expt_{FSA}^{FWEU\text{-}CMA}(\mathcal{A})$
$(sk, PK) \leftarrow FSA.Kg(1^{\kappa}, L)$, $(\overrightarrow{D}^*, \sigma^*) \leftarrow \mathcal{A}^{FASig_{sk}(\cdot), Break\text{-}in}(PK)$,
If $FSA.FAVer(PK, \overrightarrow{D}^*, \sigma^*) = 1 \wedge \forall I \subseteq \{0, \ldots, l\}, \overrightarrow{D}^* \neq ||_{k \in I} \overrightarrow{D}_k$, *return* 1, *else, return* 0.
FWEU-CMA-advantage of $\mathcal{A}$ *is* $Adv_{FSA}^{FWEU\text{-}CMA}(\mathcal{A}) = Pr[Expt_{FSA}^{FWEU\text{-}CMA}(\mathcal{A}) = 1]$.
FWEU-CMA-advantage of FSA *is* $Adv_{FSA}^{FWEU\text{-}CMA}(t, L, \mu) = \max_{\mathcal{A}}\{Adv_{FSA}^{FWEU\text{-}CMA}(\mathcal{A})\}$*, where the maximum is over all* $\mathcal{A}$ *having time complexity* t*, making at most* L *oracle queries, and the sum of lengths of these queries being at most* μ *bits.*

The above experiment does not implement a random oracle for $\mathcal{A}$ explicitly. However, we still assume the *Random Oracle Model (ROM)* [4], since Schnorr signature scheme [37] on which LogFAS is built requires the ROM.

Note that the above *FWEU-CMA* experiment and Theorem 1 also capture the *truncation attacks*:

1. The winning condition of $\mathcal{A}$ in the above experiment subsumes the truncation attack in addition to data modification. That is, $\mathcal{A}$ wins the experiment when she either modifies a data item or keeps data items intact but outputs a valid signature on a subset of a given batch query (i.e., she splits an append-only signature without knowing its individual signatures).

2. LogFAS uses a standard signature scheme SGN to *prevent truncation attacks* by computing signatures of counter values. Resilience against the traditional data forgery (without truncation) relies on EU-CMA property of $Schnorr$ and target collision-freeness of $\mathcal{IH}$. In Theorem 1, we prove that a successful truncation attack against LogFAS is equivalent to breaking SGN, and a successful data modification (including re-ordering) against LogFAS is equivalent to breaking $Schnorr$ or $\mathcal{IH}$.

4 LogFAS Schemes

In this section, we first present the intuition and detailed description of LogFAS, and then describe a LogFAS variation that has additional capabilities.

4.1 LogFAS Scheme

All existing FSA constructions [25–28,44] rely on a direct combination of an aggregate signature (e.g., [9]) and a forward-secure signature (e.g., [1,23]). Therefore, the resulting constructions simultaneously inherit all overheads of their base primitives: (i) Forward-secure signatures on individual data items, which are done separately from the append-only design, force verifiers to perform $O(l)$ ExpOps. (ii) These schemes either eliminate ExpOps from the logging phase with pre-computation but incur quadratic storage overhead to the verifiers (e.g., [44]), or require ExpOps in the logging phase for each log entry and incur linear storage overhead to the verifiers (e.g., [16,25,28]).

The above observations inspired us to design cryptographic mechanisms that can verify *the integrity of entire log entry set once directly* (preserving forward-security), instead of checking the integrity of each data item individually, though the signing operations have to be performed on individual data items. That is, instead of verifying each item one-by-one with the corresponding public key(s), verify all of them via a *single set of aggregated cryptographic components* (e.g., tokens as auxiliary signatures). These mechanisms also achieve constant storage overhead at the verifier side[3].

We achieve these goals with a provable security by using Schnorr signature and incremental hash $\mathcal{IH}$ as follows:

a) To compute a forward-secure and append-only Schnorr signature, we aggregate each individual signature s_l on D_l with the previous aggregate signature as $s_{0,l} \leftarrow s_{0,l-1} + s_l \bmod q$, $(0 < l \leq L-1, s_{0,0} = s_0)$. This is done by using a distinct private key pair (r_j, y_j) for $j = 0, \ldots, L-1$ on each data item.

b) Despite being forward-secure, the above construction still requires an ExpOp for each data item. To verify the signature on $D_0, \ldots, D_l$ with only $O(1)$ ExpOp, we introduce the notion of *token*.

In LogFAS, each Schnorr private y_j is comprised of a random key pair (a_j, d_j) for $j = 0, \ldots, L-1$. Random key a_j is mutually blinded with another random factor x_j and also a long-term private key b for $j = 0, \ldots, L-1$. The result of these blinding operations is called *auxiliary signature* (token) z_j, which can be kept publicly without revealing information about (a_j, x_j) and also can be authenticated with the long-term public key B by all verifiers. Furthermore, these masked tokens $z = z_0, \ldots, z_l$ also serve as a one-time initialization key for the incremental hash as $\mathcal{IH}_z^{q,l}$ (Definition 4), which enable verifiers to reduce the integrity of each D_j into the integrity of a final tag $z_{0,l}$. This operation preserves the integrity of each D_j and verifiability of each z_j (via public key B) without ExpOps.

c) To verify $(s_{0,l}, z_{0,l})$ via B in an aggregate form, verifiers also aggregate tokens R_j as $R_{0,l} \leftarrow \prod_{j=0}^{l} R_j \bmod p$, where p a large prime on which the group was constructed. However, initially, $(s_{0,l}, R_{0,l}, z_{0,l})$ cannot be verified directly via B, since the reduction operations introduce some extra verification information. LogFAS handles this via *auxiliary signature* (token) $M'_{0,l}$ that bridges $(s_{0,l}, R_{0,l}, z_{0,l})$ to B. That is, the signer computes an aggregate token $M'_{0,l} \leftarrow M'_{0,l-1} M_l^{e_j} \bmod p$, where $0 < l \leq L-1$ and $M_{0,0} = M_0$), along with $s_{0,l}$ in the signing process. During verification, this aggregate token eliminates the extra terms and bridges $(s_{0,l}, R_{0,l}, z_{0,l})$ with B.

This approach allows LogFAS to compute publicly verifiable signatures with only one ExpOp per-item, and this signature can be verified with only $O(1)$ ExpOps by storing only two public keys at the verifier side (regardless of the number of signers). This is much more efficient than all of its PKC-based counterparts, and also is as efficient as the symmetric schemes at the verifier side.

The detailed description of LogFAS algorithms is given below:

1) *LogFAS.Kg*$(1^\kappa, L)$: Given 1^κ, generate primes q and $p > q$ such that $q|(p-1)$, and then generate a generator α of the subgroup G of order q in $\mathbb{Z}_p^*$.

a) Generate $(b \stackrel{\$}{\leftarrow} \mathbb{Z}_q^*, B \leftarrow \alpha^{b^{-1}} \bmod p)$ and $(\widehat{sk}, \widehat{PK}) \leftarrow \mathit{SGN.Kg}(1^\kappa)$. *System-wide private key* of KGC is $\overline{sk} \leftarrow (b, \widehat{sk})$. *System-wide public key* of all verifiers is $PK \leftarrow \{p, q, \alpha, B, \widehat{PK}, L\}$.

b) Generate $(r_j, a_j, d_j, x_j) \stackrel{\$}{\leftarrow} \mathbb{Z}_q^*$ for $j = 0, \ldots, L-1$. The private key of signer ID_i is $sk \leftarrow \{r_j, y_j, z_j, M_j, R_j, \beta_j\}_{j=0}^{L-1}$, where
   - $y_j \leftarrow a_j - d_j \bmod q$, $z_j \leftarrow (a_j - x_j)b \bmod q$,
   - $R_j \leftarrow \alpha^{r_j} \bmod p$, $M_j \leftarrow \alpha^{x_j - d_j} \bmod p$,
   - $\beta_j \leftarrow \mathit{SGN.Sig}(\widehat{sk}, H(ID_i||j))$. Note that each β_j is kept secret initially, and then released as a part of a signature publicly.

[3] In all existing forward-secure and/or aggregate (append-only) logging schemes (e.g., [7,16,25,27,28,44]), the signer side storage overhead is dominated by the accumulated logs, which already incur a linear storage overhead.

2) $\mathit{LogFAS.FASig}(\langle r_l, y_l, z_l, M_l, R_l, \beta_l \rangle, D_l, \sigma_{0,l-1})$: Given $\sigma_{0,l-1}$ on $D_0, \ldots, D_{l-1}$, compute $\sigma_{0,l}$ on $D_0, \ldots, D_l$ as follows,

a) $e_l \leftarrow H(D_l||l||z_l||R_l), \;\; M'_l \leftarrow M_l^{e_l} \bmod p, \;\; s_l \leftarrow r_l - e_l y_l \bmod q$,

b) $s_{0,l} \leftarrow s_{0,l-1} + s_l \bmod q, \;\; (0 < l \leq L-1, \; s_{0,0} = s_0)$,

c) $M'_{0,l} \leftarrow M'_{0,l-1} M'_l \bmod p, \;\; (0 < l \leq L-1, \; M'_{0,0} = M_0)$,

d) $\sigma_{0,l} \leftarrow \{s_{0,l}, M'_{0,l}, \beta_l, R_j, e_j, z_j\}_{j=0}^{l}$ and erase $(r_l, y_l, s_{0,l-1}, s_l, \beta_{l-1})$.

3) $\mathit{LogFAS.FAVer}(PK, \langle D_0, \ldots, D_l \rangle, \sigma_{0,l})$:

a) If $\mathit{SGN.Ver}(\widehat{PK}, H(ID_i||l), \beta_l) = 0$ then return 0, else continue,

b) If $\prod_{j=0}^{l} R_j \bmod p = M'_{0,l} \cdot B^{z_{0,l}} \cdot \alpha^{s_{0,l}} \bmod p$ holds return 1, else return 0, where $z_{0,l} = \mathcal{IH}^{q,l}_{z_0,\ldots,z_l}(D_0||w||z_0|| R_0, \ldots, D_l||w||z_l||R_l)$.

4.2 Selective Verification with LogFAS

All the previous FSA constructions (e.g., [25–27, 44, 45]) verify the set of log entries via only the final aggregate signature to prevent the truncation attack and save the storage. However, this approach causes performance drawbacks: (i) The verification of any subset of log entries requires the verification of the entire set of log entries (i.e., always $O(L)$ ExpOps for the subset verification). (ii) The failure of signature verification does not give any information about which log entries were corrupted.

Ma et al. proposed immutable-FssAgg (iFssAgg) schemes in [28] to allow fine-grained verification without being vulnerable to truncation attacks. However, iFssAgg schemes double the signing/verifying costs of their base schemes. In addition, even if the signature verification fails due to only a few corrupted log entries (i.e., accidentally damaged entry(ies)), detecting which log entry(ies) is (are) responsible for the failure requires verifying each individual signature.

LogFAS can address the above problems via a simple variation without incurring any additional costs: The signer keeps *all* signatures and tokens in their individual forms (including s_j for $j = 0, \ldots, l$) without aggregation. The verifiers can aggregate them according to their needs by preserving the security and verifiability. This offers performance advantages over iFssAgg schemes [28]:

(i) LogFAS protects the number of log entries via pre-computed tokens $\beta_0, \ldots, \beta_l$, and therefore individual signatures can be kept without a truncation risk. This eliminates the necessity of costly immutability strategies used in iFssAgg schemes [28]. Furthermore, a verifier can selectively aggregate any subset of $l' < l$ log entries and verify them by performing only $O(1)$ ExpOps as in the original LogFAS. This is much more efficient than the iFssAgg schemes, which require $O(2l')$ ExpOps.

(ii) LogFAS can use a recursive subset search strategy to identify corrupted log entries causing the verification failure faster than linear search[4]. That is, the set of log entries is divided into subsets along with their corresponding individual signatures. Each subset is then independently verified by $\mathit{LogFAS.AVer}$ via its corresponding aggregate signature, which is efficiently computed from individual signatures. Subsets returning 1 are eliminated from the search, while each subset returning 0 is again divided into subsets and verified by $\mathit{LogFAS.AVer}$ as described. This subset search continues recursively until all the corrupted log entries are identified.

The above strategy can quickly identify the corrupted entries when most log entries are intact. For instance, if only one entry is corrupted, it can identify the corrupted entry by performing $(2 \log_2 l)$ ExpOps + $O(l)$ hash operations. This is much faster than linear search used in the previous PKC-based schemes, which always requires $O(l)$ ExpOps + $O(l)$ hash operations.

Recursive subset strategy remains more efficient than linear search as long as the number of corrupted entries c satisfies $c \leq \frac{l}{2 \log_2 l}$. When $c > \frac{l}{2 \log_2 l}$, depending on c and the distribution of corrupted entries, recursive subset search might be more costly than linear search. To minimize the performance loss in such an inefficient

[4] Note that the previous PKC-based audit logging schemes *cannot* use such a recursive subset search strategy to identify corrupted log entries with a sub-linear number ExpOps, since they always require linear number of ExpOps to verify a given subset from the entire log entry set (in contrast to LogFAS that requires $O(1)$ExpOp to verify a given subset).

case, the verifier can switch from recursive subset search to the linear search if the recursive division and search step continuously returns 0 for each verified subset. The verifier can ensure that the performance loss due to an inefficient case does not exceed the average gain of an efficient case by setting the maximum number of recursive steps to be executed to $l'/2 - \log_2 l'$ for each subset with l' entries.

5 Security Analysis

We prove that LogFAS is a *FWEU-CMA* signature scheme in Theorem 1 below.

Theorem 1 $Adv^{\mathit{FWEU\text{-}CMA}}_{\mathit{LogFAS}}(t, L, \mu)$ *is bounded as follows,*

$$Adv^{\mathit{FWEU\text{-}CMA}}_{\mathit{LogFAS}}(t, L, \mu) \quad \leq \quad L \ \cdot \ Adv^{\mathit{EU\text{-}CMA}}_{\mathit{Schnorr}}(t', 1, \mu') \ + \ Adv^{\mathit{EU\text{-}CMA}}_{\mathit{SGN}}(t'', L, \mu'') \ + \ Adv^{\mathit{TCR}}_{\mathcal{IH}^{q,L}_{z}}(t'''),$$

where $t' = O(t) + L \cdot O(\kappa^3)$ *and* $\mu' = \mu/L$.

Proof: Let $\mathcal{A}$ be a LogFAS attacker. We construct a *Schnorr* attacker $\mathcal{F}$ that uses $\mathcal{A}$ as a sub-routine as follows:

Algorithm $F^{\mathit{Schnorr.Sig}_y(\cdot)}(Y)$

Set the target forgery index $w \xleftarrow{\$} [0, L-1]$,

$(sk, PK) \leftarrow \mathit{LogFAS.Kg}(1^\kappa, L)$, where $sk = \{\langle b, \overline{sk}\rangle, \langle ID_i : r_j, y_j, z_j, M_j, R_j, \beta_j\rangle\}_{j=0}^{L-1}$ and $PK = (p, q, \alpha, B, \widehat{PK}, L)$,

To embed Schnorr public key Y into token M_w, simulate tokens (M_w, z_w) as follows:

-$(\gamma, \gamma') \xleftarrow{\$} \mathbb{Z}_q^*,\ M_w \leftarrow Y \cdot \alpha^{(-\gamma + \gamma' b^{-1})} \bmod p,\ z_w \leftarrow \gamma \cdot b - \gamma' \bmod q$,

Execute $\mathcal{A}^{\mathit{FASig}_{sk}(\cdot), \mathit{Break\text{-}in}}(PK)$ as follows:

- $l \leftarrow 0,\ j' \leftarrow 0,\ i \leftarrow 0$,
- Queries: $\mathcal{A}$ first queries $\mathit{FASig}_{sk}(\cdot)$ and then *Break-in* oracles up to L messages of her choice in total:

  • How to respond i-th $\mathit{FASig}_{sk}(\cdot)$ query:

  - For each query i, $\mathcal{A}$ queries $\mathit{FASig}_{sk}(\cdot)$ on $\overrightarrow{D}_i = \{D_{j'}, \ldots, D_j\},\ j > j'$ of her choice. If $j + 1 > L$ then reject the query and proceed to the *Forgery phase* ($\mathcal{A}$ exceeds her query limit). Otherwise, continue to the next step,
  - If $j' \leq w \leq j$ then $\mathit{FASig}_{sk}(\cdot)$ goes to the Schnorr oracle on D_w as $(s_w, R_w, e_w) \leftarrow \mathit{Schnorr.Sig}_y(D_w || w || z_w)$. $\mathit{FASig}_{sk}(\cdot)$ then computes $s_{j',j} \leftarrow \sum_{j' \leq k \leq j, k \neq w} (r_k - e_k y_k) + s_w \bmod q$, where $e_k \leftarrow H(D_k || k || z_k || R_k)$ for $k = j', \ldots, j$. Also set variable $D' \leftarrow D_w$. Otherwise, compute $s_{j',j} \leftarrow \sum_{k=j'}^{j} (r_k - e_k y_k) \bmod q$, where $e_k \leftarrow H(D_k || k || z_k || R_k)$ for $k = j', \ldots, j$,
  - $M'_{j',j} \leftarrow \prod_{k=j'}^{j} M_k^{e_k} \bmod p$,
  - $s_{0,j} \leftarrow s_{0,j'-1} + s_{j',j} \bmod q$ and $M'_{0,j} \leftarrow M'_{0,j'-1} M'_{j',j} \bmod p$, where $(s_{0,j'-1}, M'_{0,j'-1})$ were computed on $\mathcal{A}$'s previous queries $D_0, \ldots, D_{j'-1}$, (for initial $j' = 0$, $s_{0,j'-1} = 0, M_{0,j'-1} = 1$),
  - Response i-th query of $\mathcal{A}$ as $\sigma_{0,j} \leftarrow \{s_{0,j}, M'_{0,j}, \beta_j, R_k, e_k, z_k\}_{k=j'}^{j}$,
  - $\mathcal{F}$ maintains four lists and a variable z' in addition to D' for bookkeeping purposes. Insert i-th query $\overrightarrow{D}_i$ into data list $\mathcal{LD}[i]$. Insert signature results $(s_{0,j}, M_{0,j}, \beta_j)$ into the lists $(\mathcal{LS}1, \mathcal{LS}2, \mathcal{LS}3)$, respectively. Also update variable z' as $z' \leftarrow \mathcal{IH}_z^{q,j}(e_0, \ldots, e_j)$ for $z = z_0, \ldots, z_j$,
  - If $\mathcal{A}$ decides to the “break-in”, proceed to the next step. Otherwise, update $j' \leftarrow j+1,\ l \leftarrow j',\ i \leftarrow i+1$ and continue to respond her queries,

  • How to respond queries to the *Break-in* oracle: $\mathcal{A}$ queried l individual messages to $\mathit{FASig}_{sk}(\cdot)$ oracle up to now. If $l = L$ then reject the query (all private keys were used and erased) and proceed to the next step. Otherwise, if $l < w$ then *abort* and return 0 ($\mathcal{F}$ does not know the corresponding Schnorr private key index w). Otherwise, supply $\mathcal{A}$ with $\{r_j, y_j, z_j, M_j, R_j, \beta_j\}_{j=l+1}^{L-1}$.

- Forgery: Finally, $\mathcal{A}$ outputs a forgery as $(\langle D_0^*, \ldots, D_k^* \rangle, \sigma^*)$, where $\sigma^* = \{s_{0,k}^*, M_{0,k}^*, \beta^*, R_j^*, e_j^*, z_j^*\}_{j=0}^{k}$.

By Definition 7, $\mathcal{A}$ wins if the following condition holds:

1. $LogFAS.FAVer(PK, \langle D_0^*, \ldots, D_k^* \rangle, \sigma^*) = 1$

2. $\forall I \subseteq \{0, \ldots, i\}, \{D_0^*, \ldots, D_k^*\} \neq ||_{m \in I} \mathcal{LD}[m]$,

If one of the above conditions fails, $\mathcal{A}$ loses in *FWEU-CMA* experiment, and therefore $\mathcal{F}$ *aborts* and returns 0. Otherwise, $\mathcal{F}$ proceeds according to one of the following cases, which are implied by condition 2 as follows:

a) $\exists j \in \{0, \ldots, k\} : (D_j^* \notin \{\mathcal{LD}[0], \ldots, \mathcal{LD}[i]\} \wedge k = L - 1)$

b) $\exists j \in \{0, \ldots, i\} : (\{D_0^*, \ldots, D_k^*\} \subset \mathcal{LD}[j] \ \wedge \ \beta^* \notin \mathcal{LS}3)$

c) $\exists j \in \{0, \ldots, k\} : (D_j^* \notin \{\mathcal{LD}[0], \ldots, \mathcal{LD}[i]\} \ \wedge \ z' = \mathcal{IH}_z^{q,k}(D_0^*, \ldots, D_k^*))$

Case a): This case implies $\mathcal{A}$ modifies at least one data item (without truncation). $\mathcal{F}$ checks if $D_w^* \neq D'$ (i.e., whether one of $\mathcal{A}$'s forgery is on D', whose corresponding token includes Schnorr public key Y that $\mathcal{F}$ embedded). If it fails, $\mathcal{F}$ *aborts* and return 0. Otherwise, by Definition 2, $\mathcal{F}$ wins the *EU-CMA* experiment and returns 1, since the conditions below hold:

$Schnorr.Ver(\langle p, q, \alpha, Y \rangle, D_w^*, \langle s_w^*, R_w^*, e_w^* \rangle) = 1$ and $\mathcal{F}$ did not ask D_w^* to the Schnorr oracle, where $s_w^* \leftarrow s_{0,L-1}^* - \sum_{0 \leq m \leq L-1, m \neq w}^{L-1} (r_m - e_m^* y_m) \bmod q$ and $e_m^* \leftarrow H(D_m^* || m || z_m^* || R_m^*)$ for $m = 0, \ldots, L-1$.

Since the target forgery index is randomly chosen as $w \overset{\$}{\leftarrow} [0, L-1]$, if $\mathcal{A}$ wins the experiment based on this case with the probability $Adv_{LogFAS}^{FWEU\text{-}CMA}(t, L, \mu)$, then $\mathcal{F}$ wins with the probability $Adv_{LogFAS}^{FWEU\text{-}CMA}(t, L, \mu)/L$.

The running time of $\mathcal{F}$ is that of $\mathcal{A}$ plus the overhead due to handling $\mathcal{A}$'s queries as $t' = O(t) + L \cdot O(\kappa^3)$, where $O(\kappa^3)$ denotes the execution time of modular exponentiation operation in Z_p^* for given κ.

Case b): This case implies a successful tail-truncation attack. If it holds, then by Definition 2, $\mathcal{A}$ breaks SGN since β^* is valid and it was not queried. This happens with probability $Adv_{SGN}^{EU\text{-}CMA}(t'', L, \mu'')$.

Case c): If this case holds, then by Definition 5, $\mathcal{A}$ breaks $\mathcal{IH}$ by finding a target collision. This happens with probability $Adv_{\mathcal{IH}_z^{q,L}}^{TCR}(t''')$.

In the above experiment, the simulated view of $\mathcal{A}$ is *perfectly indistinguishable* from the real view of $\mathcal{A}$: The real view of $\mathcal{A}$ after L queries ($0 \leq l \leq L$ queries to the $FASig_{sk}(\cdot)$ oracle and $L - l$ queries to the *Break-in* oracle) is $\overrightarrow{A}_{Real} = \{PK, \mathcal{LS}1, \mathcal{LS}2, \mathcal{LS}3, e_i, R_j, z_j, M_{l+1}, \ldots, M_{L-1}\}_{i=0,j=0}^{l,L-1}$, where all keys/tokens/signatures are computed/generated via original LogFAS algorithms. The simulated view of $\mathcal{A}$ after L queries is equivalent to $\overrightarrow{A}_{Real}$ except that (M_w, z_w) are simulated as described. One may verify that the joint probability distribution of these views are identical as $Pr[\overrightarrow{A}_{Real} = \overrightarrow{a}] = Pr[\overrightarrow{A}_{Sim} = \overrightarrow{a}]$. □

Remark 1 Another security concern in audit logging is *delayed detection* identified in [27]. In delayed detection, log verifiers cannot detect whether the log entries are modified until an online TTP provides auxiliary keying information to them. LogFAS does not rely on an online TTP support or time factor to achieve the signature verification, and therefore it is not prone to delayed detection.

6 Performance Analysis and Comparison

In this section, we present the performance analysis of LogFAS and compare it with previous schemes. We follow the notation in Table 2 in our analysis and comparison.

Computational Overhead: In LogFAS, the costs of signing a single item is $Exp + Mul + H$ including the key update cost. The cost of verifying l items is $2Exp + O(l(3Mul + H))$. The key generation cost for L items is $O(L(2Exp + Mul))$.

Table 3 and Table 4 compare the computational cost of LogFAS with its counterparts analytically and numerically, respectively.

From a verifier's perspective, LogFAS requires only a small and constant number of Exp operations regardless of the number of log entries to be verified. Therefore, it is much more efficient than all PKC-based schemes,

Table 2: Notation used in performance analysis and comparison of LogFAS and its counterparts

$GKg/GSig/GVer$: Generic key gen/sig/verifying ops.	Sqr: Squaring mod n'	H: Hashing
$Mul/Mulq'/Mulp'/Muln'$: Mul. mod p, q', p', n'	$Exp/Expp'$: Exp. mod p and p'	L: Max. # of key upd.
w and l:# of data items processed and will be processed	l': # data items to be processed in a subset	S: # of signers
$Add/Addq'$: Add. mod q and q', resp.	PR: pairing op.	z: FssAgg sec. param.

Suggested bit lengths to achieve 80-bit security for the above parameters are as follows for each compared scheme (based on the parameters suggested by Lenstra et al. in [24]): Large primes ($|p| = 2048, |q| = 1600$) for LogFAS and Logcrypt, primes ($|p'| = 512, |q'| = 160$) for BAF and FssAgg-BLS, ($|n'| = 1024, z = 160$) for FssAgg-AR and FssAgg-BM, where n' is Blum-Williams integer [25].

Table 3: Computation involved in LogFAS and its counterparts

		Online		**Offline**
		ASig & Upd (per item)	**AVer** (l or l' entries)	**KG** (max. L)
PKC-based	*LogFAS*	$Exp + Mul + H$	$2Exp + O(l(3Mul + H))$	$O(L(2Exp + Mul))$
	FssAgg-BLS	$MtP + Expp' + Mulp'$	$l(Mulp' + H + PR)$	$O(L(H + Expp'))$
	FssAgg-BM	$\frac{z}{2}Muln' + z \cdot Sqr$	$O(L \cdot Sqr) + \frac{l \cdot z}{2}Muln'$	$O(L(z \cdot Sqr + \frac{z}{2}Muln'))$
	FssAgg-AR	$3z \cdot Sqr + \frac{z}{2}Muln'$	$z(L + l)Sqr + (2l + l \cdot z)Muln'$	$O(2L(z \cdot Sqr + \frac{z}{2}Muln'))$
	iFssAgg	$2 \cdot ASig + Upd$	$2 \cdot AVer(L, l')$	Kg
	Logcrypt	$GSig$	$O(l \cdot GVer)$	$O(L \cdot GKg)$
	BAF	$3H + Mulq' + 2Addq'$	$O(2l(EMul + H))$	$O(2L(H + EMul))$
Symmetric		H	$O(l \cdot H)$	$O(L \cdot H)$

LogFAS is the only scheme achieving verification with $O(1)ExpOp$ regardless of the value of (l, l'), while their counterparts require either $O(l)ExpOp$ (FssAgg, Logcrpyt and BAF) or $O(l')ExpOp$ (iFssAgg schemes). At the same time, LogFAS is as efficient as their counterparts at the signer side except the BAF.

which require one ExpOp per log entry. Besides, it does not double the verification cost to prevent the truncation attacks, providing further efficiency over iFssAgg schemes [28]. For instance, the verification of 10,000 log entries with LogFAS is 2650, 479, 1937, 1427 and 208 times faster than that of FssAgg-BLS, FssAgg-BM, FssAgg-AR, Logcrypt, and BAF, respectively. The verification of subsets from these entries with LogFAS is also much more efficient than all of its counterparts as shown in Table 4. The execution time differences with LogFAS and its PKC-based counterparts grow linearly with respect to the number of log entries to be verified. Initially, the symmetric schemes are more efficient than all PKC-based schemes, including ours. However, since the verification operations of LogFAS are dominated by H, their efficiency become comparable with symmetric schemes as the number of log entries increases (e.g., $l = 10^4$). From a logger's perspective, LogFAS is also more efficient than its PKC-based counterparts with the exception of BAF.

All PKC-based schemes require $O(L)$ ExpOps in the key generation phase.

Signature/Key/Data Storage and Transmission Overheads: LogFAS is a verifier storage friendly scheme; it requires each verifier to store only two public keys and an index along with system-wide parameters (e.g., $|q| + |4p|$), regardless of the number of signers or the number of log entries to be verified.

In LogFAS, the append-only signature size is $|q|$. The key/token and data storage overheads on the logger side are linear as (i.e., $O(L(5|q| + 2|p|)) + O(l|D|)$) (assuming SGN is chosen as Schnorr [37]). LogFAS transmits a token set along with each data item requiring $O(l(|q| + |p| + |D|))$ transmission in total. The fine-grain verification introduces $O(l')$ extra storage/communication overhead due to the individual signatures.

From a verifier's perspective, LogFAS is much more storage efficient than all existing schemes, which require either $O(L \cdot S)$ (e.g., FssAgg-BLS [26] and BAF [44]), or $O(S)$ (e.g., [6, 7, 16, 25, 28, 35, 36]) storage. From a logger's perspective, all the compared schemes both accumulate (stores) and transmit linear number of data items (i.e., $O(l)D$) until their verifiers become available to them. This dominates the main storage and communication overhead for these schemes. In addition to this, LogFAS requires linear key storage overhead at the logger side, which is slightly less efficient than [25, 26, 44]. LogFAS with fine-grained verification and its counterpart iFssAgg schemes [28] both require linear key/signature/data storage/transmission overhead.

Availability, Applicability and Security: The symmetric schemes [6, 7, 35, 36] are not publicly verifiable and

Table 4: Execution time (in ms) comparison of LogFAS and its counterparts

Criteria			*PKC-based*						*Sym.*
			LogFAS	FssAgg (l) / iFssAgg (l')			Logcrypt	BAF	
			($l = 10^4, l' < l$)	BLS / i	BM / i	AR / i			
Off.	*Kg,* $L = 10^4$		5.06×10^4	3.3×10^3	8.8×10^4	1.7×10^5	2.6×10^4	4×10^4	$\tilde{2}0$
Onl.	*Sig&Upd (1)*		1.2	1.8 / 3.6	13.1 / 26.2	28 / 56	2.05	0.007	0.004
	Ver.	$l' = 10^2$	72.87	4.8×10^3	1.8×10^3	1.6×10^5	1.4×10^3	0.2×10^3	0.2
		$l' = 10^3$	75.2	4.8×10^4	1×10^4	1.8×10^5	1.5×10^4	2.05×10^3	2
		$l = 10^4$	98.12	2.6×10^5	4.7×10^4	1.9×10^5	1.4×10^5	2.04×10^4	19.9

(i) The execution times were measured on a computer with an Intel(R) Xeon(R)-E5450 3GHz CPU and 4GB RAM running Ubuntu 9.04. We tested LogFAS, BAF [44], FssAgg-BLS [26], Logcrypt (with DSA), and the symmetric schemes (e.g., [7,26,36]) using the MIRACL library [38], and FssAgg-AR/BM using the NTL library [39]. Parameter sizes determining the execution times of each scheme were selected s.t. $\kappa = 80$ (parameter sizes were discussed in Table 2).

also require online server support to verify log entries. Furthermore, they are vulnerable to both truncation and delayed detection attacks [27,28] with the exception of FssAgg-MAC [26]. In contrast, PKC-based schemes [16, 25–28] are publicly verifiable without requiring online server support, and they are secure against the truncation and delayed detection attacks, with the exception of Logcrypt [16].

LogFAS achieves all the desirable availability/applicability and security properties as well as being significantly more efficient than PKC-based schemes.

7 Related Work

Most closely related are those forward-secure audit logging schemes [6, 7, 16, 25–28, 35, 36, 44]. The detailed comparison of these schemes with LogFAS has been presented in Section 6.

Apart from the above schemes, there is a set of works complementary to ours. Itkis [19] proposed cryptographic tamper resistance techniques that can detect tampering even if all the keying material is compromised. LogFAS can be combined with Itkis model as any forward-secure signature [19]. Yavuz et al. [45] proposed a Hash-based Forward-Secure and Aggregate Signature Scheme (HaSAFSS) for unattended wireless sensor networks, which uses timed-release encryption to achieve computational efficiency. Waters et al. proposed an audit logging scheme [42] relying on Identity-Based Encryption (IBE), which enables encrypted search on the log entries. Davis et al. proposed time-scoped search techniques on encrypted audit logs [13]. These schemes can be coupled with LogFAS to provide confidentiality. There are also authenticated data structures that can be used for audit logging in distributed systems [12,33]. LogFAS can serve as a digital signature primitive needed by these constructions.

Chong et al. proposed an extension to the scheme in [35] by using tamper-resistant hardware [11]. Xu et al. proposed SAWS [43] to build a Trusted Computing Base (TCB) to protect private keys used in PKC operations.

8 Conclusion

In this paper, we proposed a new forward-secure and append-only audit logging scheme called LogFAS. LogFAS achieves public verifiability without requiring any online trusted server support, and is secure against truncation and delayed detection attacks. LogFAS is much more computationally efficient than all existing PKC-based alternatives, with a performance comparable to symmetric schemes at the verifier side. LogFAS is also the most verifier storage efficient scheme among all existing alternatives. Last, a variation of LogFAS enables selective subset verification and efficient search of corrupted log entries. Overall, our comparison with the existing schemes shows that LogFAS is an ideal choice for secure audit logging by offering high efficiency, security, and public verifiability simultaneously for real-life applications.

Acknowledgment

We would like to thank Dr. Di Ma who kindly provided her implementation of FssAgg schemes [25,26,28].

References

[1] M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. In *Advances in Crpytology (ASIACRYPT '00)*, pages 116–129. Springer-Verlag, 2000.

[2] R. Anderson. Two remarks on public-key cryptology, invited lecture. Proceedings of the 4th ACM conference on Computer and Communications Security (CCS '97), 1997.

[3] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proc. of the 16th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '97)*, pages 163–192. Springer-Verlag, 1997.

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and Communications Security (CCS '93)*, pages 62–73, NY, USA, 1993. ACM.

[5] M. Bellare and P. Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In *Proceedings of Advances in Cryptology (CRYPTO '97)*, pages 470–484, London, UK, 1997. Springer-Verlag.

[6] M. Bellare and B. S. Yee. Forward integrity for secure audit logs, 1997.

[7] M. Bellare and B. S. Yee. Forward-security in private-key cryptography. In *Proceedings of the The Cryptographers Track at the RSA Conference (CT-RSA '03)*, pages 1–18, 2003.

[8] J. Bethencourt, D. Boneh, and B. Waters. Cryptographic methods for storing ballots on a voting machine. In *Proc. of the Network and Distributed System Security Symposium (NDSS 07')*, 2007.

[9] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proc. of the 22th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '03)*, pages 416–432. Springer-Verlag, 2003.

[10] E. C. E. Centera. `http://www.emc.com/products/family/emc-centera-family.htm`.

[11] C. N. Chong and Z. Peng. Secure audit logging with tamper-resistant hardware. In *Proceedings of the 18th IFIP International Information Security Conference*, pages 73–84. Kluwer Academic Publishers, 2003.

[12] S. Crosby and D. S. Wallach. Efficient data structures for tamper evident logging. In *Proceedings of the 18th conference on USENIX Security Symposium*, August 2009.

[13] D. Davis, F. Monrose, and M. Reiter. Time-scoped searching of encrypted audit logs. In *Proc. of the 6th International Conference on Information and Communications Security (ICICS '04)*, pages 532–545, 2004.

[14] I. C. I. T. DR550. `http://www.ibm.com/servers/storage/disk/dr`.

[15] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 9th conference on Applications, technologies, architectures, and protocols for computer communications, (SIGCOMM '03)*, pages 27–34. ACM, 2003.

[16] J. E. Holt. Logcrypt: Forward security and public verification for secure audit logs. In *Proc. of the 4th Australasian workshops on Grid computing and e-research (ACSW '06)*, pages 203–211, 2006.

[17] W. W. Hsu and S. Ong. Technical forum: WORM storage is not enough. *IBM System Journal*, 46(2):363–369, 2007.

[18] R. Impagliazzo and M. Naor. Efficient cryptographic schemes provably as secure as subset sum. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 236–241, Washington, DC, USA, 1989. IEEE Computer Society.

[19] G. Itkis. Cryptographic tamper evidence. In *Proc. of the 10th ACM conference on Computer and communications security (CCS '03)*, pages 355–364, New York, NY, USA, 2003. ACM.

[20] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *Advances in Cryptology (CRYPTO '01)*, pages 332–354. Springer-Verlag, 2001.

[21] N. Koblitz and A. Menezes. Pairing-based cryptography at high security levels. Cryptology ePrint Archive: Report 2005/076, 2005.

[22] A. Kozlov and L. Reyzin. Forward-secure signatures with fast key update. In *Proc. of the 3rd International Conference on Security in Communication Networks (SCN '02)*, 2002.

[23] H. Krawczyk. Simple forward-secure signatures from any signature scheme. In *Proceedings of the 7th ACM conference on Computer and Communications Security, (CCS '00)*, pages 108–115. ACM, 2000.

[24] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[25] D. Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 3rd ACM symposium on Information, Computer and Communications Security (ASIACCS '08)*, pages 341–352, NY, USA, 2008. ACM.

[26] D. Ma and G. Tsudik. Forward-secure sequential aggregate authentication. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (S&P '07)*, pages 86–91, May 2007.

[27] D. Ma and G. Tsudik. A new approach to secure logging. In *Proc. of the 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSEC '08)*, pages 48–63, 2008.

[28] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transaction on Storage (TOS)*, 5(1):1–21, 2009.

[29] T. Malkin, D. Micciancio, and S. K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Proc. of the 21th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '02)*, pages 400–417. Springer-Verlag, 2002.

[30] Y. Mu, W. Susilo, and H. Zhu. Compact sequential aggregate signatures. In *Proceedings of the 22nd ACM symposium on Applied computing (SAC '07)*, pages 249–253. ACM, 2007.

[31] D. Naccache, D. M'Raïhi, S. Vaudenay, and D. Raphaeli. Can D.S.A. be improved? complexity tradeoffs with the digital signature standard. In *Proc. of the 13th International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT '94)*, pages 77–85, 1994.

[32] A. Oprea and K. D. Bowers. Authentic time-stamps for archival storage. In *14th European Symposium on Research in Computer Security, (ESORICS '09)*, pages 136–151, Berlin, Heidelberg, 2009. Springer-Verlag.

[33] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. of the 15th ACM conference on Computer and Communications Security (CCS 2008)*, pages 437–448, New York, NY, USA, 2008. ACM.

[34] D. Pointcheval and J. Stern. Security proofs for signature schemes. In *Proc. of the 15th International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT '96)*, pages 387–398. Springer-Verlag, 1996.

[35] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. of the 7th conference on USENIX Security Symposium*. USENIX Association, 1998.

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

[37] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[38] Shamus. Multiprecision integer and rational arithmetic c/c++ library (MIRACL). `http://www.shamus.ie/`.

[39] V. Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

[40] D. Stinson. *Cryptography: Theory and Practice,Second Edition*. CRC/C&H, 2002.

[41] Y. Wang and Y. Zheng. Fast and secure magnetic worm storage systems. In *Proc. of the 2nd IEEE International Security in Storage Workshop (SISW '03)*, pages 11–25, October 2003.

[42] B. Waters, D., G. Durfee, and D. Smetters. Building an encrypted and searchable audit log. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '04)*, 2004.

[43] W. Xu, D. Chadwick, J. Li, and S. Otenko. A PKI-based secure audit web service. In *Proc. IASTED Int. Conf. on Communication, Network, and Information Security (CNIS '05)*, 2005.

[44] A. A. Yavuz and P. Ning. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proceedings of 25th Annual Computer Security Applications Conference (ACSAC '09)*, pages 219–228, 2009.

[45] A. A. Yavuz and P. Ning. Hash-based sequential aggregate and forward secure signature for unattended wireless sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous '09)*, July 2009.