BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems

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

Abstract—**Audit logs, providing information about the current and past states of systems, are one of the most important parts of modern computer systems. Providing security for audit logs on an untrusted machine in a large distributed system is a challenging task, especially in the presence of active adversaries. In such a system, it is critical to have *forward security* such that when an adversary compromises a machine, she cannot modify or forge the log entries accumulated before the compromise. Unfortunately, existing secure audit logging schemes have significant limitations that make them impractical for real-life applications: Existing Public Key Cryptography (PKC) based schemes are computationally expensive for logging in task intensive or resource-constrained systems, while existing symmetric schemes are not publicly verifiable and incur significant storage and communication overheads.**

In this paper, we propose a novel forward secure and aggregate logging scheme called *Blind-Aggregate-Forward (BAF)* logging scheme, which is suitable for large distributed systems. BAF can produce publicly verifiable forward secure and aggregate signatures with near-zero computational, storage, and communication costs for the loggers, without requiring any online Trusted Third Party (TTP) support. We prove that BAF is secure under appropriate computational assumptions, and demonstrate that BAF is significantly more efficient and scalable than the previous schemes. Therefore, BAF is an ideal solution for secure logging in both task intensive and resource-constrained systems.

Keywords-**Applied cryptography; secure audit logging; digital forensics; forward security; signature aggregation.**

I. Introduction

Audit logs are a fundamental digital forensic mechanism for providing security in computer systems. They are used to keep track of important events about the system activities such as program executions/crashes, data modifications, and user activities. Providing information about the current and past states of systems, audit logs are invaluable parts of system security. The forensic value of audit logs makes them an attractive target for attackers, who aim to erase the traces of their malicious activities recorded by logs. Indeed, the first target of an experienced attacker is generally the audit logs [1], [2].

Some naive audit log protection techniques include using a bug-free tamper-resistant hardware (to prevent the attacker from reaching audit logs), and maintaining a continuous and secure communication channel between each logger and remote trusted entity(ies) (to upload logs to a trusted entity in real-time before the attack occurs). However, as pointed out by some recent studies (e.g., [1], [3]–[5]), these techniques are impractical for modern computer systems. In large distributed systems (e.g., virtual computing cloud), it is impractical to assume a continuous end-to-end real-time communication between a trusted center and a logger [6]. Similarly, assuming a tamper-resistant hardware being "bug free" and guaranteeing its presence on all types of platforms are equally impractical (e.g., logging in smart cards, implantable devices [7] and wireless sensors [8]).

To address the above problems, a set of cryptographic countermeasures have been proposed to enable secure logging on untrusted machines, without assuming a tamper-resistant hardware or continuous real-time log verifiers (e.g., [1]–[3], [5], [9]). In the setting where there is no tamper resistant hardware nor continuous real-time communication, the untrusted machine has to accumulate audit log entries when the log verifiers are not available. After the attacker compromises the system, no cryptographic technique can prevent her from manipulating the post-attack log entries (due to her control over the system). However, it is critical to prevent the attacker from manipulating the log entries previously accumulated before the compromise. Such a security property is referred to as *forward security* [1], [3], [4].

One group of these schemes rely on symmetric cryptography to provide forward security in a computationally efficient way by using forward-secure Message Authentication Codes (MACs), Pseudo Random Number Generators (PRNGs) (e.g., [4], [5], [10]), and one-way hash chains (e.g., [2], [5], [8]). Despite their simplicity and computational efficiency, these schemes have significant limitations: (i) Due to their symmetric nature, these schemes cannot achieve public verifiability. As a result, they either require full symmetric key distribution (e.g., FssAgg-MAC [8]) or online TTP support (e.g., [2], [4], [5], [10]). While full symmetric key distribution incurs significant storage overhead to system entities, the online TTP requirement brings architectural difficulties, increases communication overhead, and makes the system vulnerable to certain attacks (e.g., truncation and delayed detection attacks [1], [3]). (ii) All the above schemes incur high storage and communication overheads to the loggers, since they require storing and transmitting an authentication tag for each log entry (or logging period) (e.g., [2], [4], [5], [10]).

The other group of schemes rely on Public Key Cryptography (PKC). The scheme in [11] extends the forward-secure MAC strategy to the PKC domain to achieve public verifiability. However, it still incurs significant storage and communication overheads due to the requirements of storing and transmitting an authentication tag for each log entry or logging period. Recently, a series of studies have been proposed to reduce storage and communication overheads via forward secure and aggregate schemes (e.g., FssAgg-BLS [8], FssAgg-BM and FssAgg-AR [1], [3], [9]). These schemes require only one authentication tag for all the accumulated log entries (due to the ability of aggregating individual authentication tags into a single compact tag), and therefore are storage and bandwidth efficient. Unfortunately, all these PKC-based schemes are computationally expensive for the logger and even more for the log verifier. These costs make them impractical for logging in task-intensive or resource constrained systems.

The above discussion indicates that an efficient audit logging mechanism refrained from all the above limitations is solely needed. In order to fulfill this requirement, we propose a novel forward secure and aggregate logging scheme for secure audit logging in distributed systems, which we call *Blind-Aggregate-Forward (BAF)* logging scheme. BAF can address all the aforementioned limitations of the existing approaches simultaneously. We summarize the properties of BAF below:

1) *Efficient Log Generation:* In BAF, the computational cost of logging a single data item is only three cryptographic hash operations. This is as efficient as existing symmetric schemes (e.g., [2], [4], [5], [8], [10]), and is much more efficient than all existing PKC-based schemes (e.g., [1], [3], [8], [9], [11]).
2) *Logger Storage/Bandwidth Efficiency:* BAF introduces *near zero* storage and communication overheads to the logger. That is, independent from the number of time periods or data items to be signed, the storage overhead of the logger is always *constant* and so is the size of the resulting signature (which is equal to a single compact signature). Thus, our scheme is much more storage/bandwidth efficient than existing symmetric schemes (e.g., linear overhead on the logger [2], [4], [5], [8], [10]).
3) *Efficient Log Verification:* In BAF, the computational cost of verifying a single log entry is only a single ECC scalar multiplication, which is more efficient than existing PKC-based schemes (e.g., [1], [3], [8], [9], [11]).
4) *Public Verifiability:* BAF produces publicly verifiable signatures (which implies no full symmetric key distribution), and therefore is much more scalable for distributed systems than symmetric schemes (e.g., [2], [4], [5], [8], [10]).
5) *Offline TTP and Immediate Verification:* Unlike some previous schemes (e.g., [2], [5]), BAF does not need online TTP support to enable log verification. Hence, it eliminates the bandwidth overhead that stems from the frequent communication between log verifiers and the TTP. This also makes BAF more scalable and reliable due to the simple architectural design and being free of single point of failures. Last, since BAF achieves immediate verification, it is secure to delayed detection attacks [1][1].

The above properties make BAF a perfect choice for secure audit logging in large distributed systems even for highly resource constrained environments such as smart cards, implantable devices [7] and wireless sensors [8].

The remainder of this paper is organized as follows. Section II provides the BAF syntax and security model. Section III describes BAF in detail. Section IV gives detailed security analysis of BAF. Section V presents performance analysis and compares BAF with previous approaches. Section VI briefly discusses the related work. Section VII concludes this paper.

II. Syntax and Security Model

We first give notation and assumptions used in our scheme. We then give the BAF *syntax* to clarify generic BAF algorithms, following the example of [8], [9], [12], [13]. Such a syntax enables us to formally define the *security model*, in which BAF is analyzed for the forward secure and aggregate unforgeability against adaptive chosen plaintext attacks. The actual algorithms are presented in Section III, while the analysis of BAF security model is provided in Section IV.

A. Syntax

Notation. G is a generator of group $\mathbb{G}$ defined on an Elliptic Curve (EC) $E(F_p)$ over a prime field F_p, where p is a large prime number and q is the order of G. kG, where k is an integer, denotes a *scalar multiplication*. $x \overset{R}{\leftarrow} F_p$ denotes that x is selected uniformly from F_p. Operators $||$ and $|x|$ denote the concatenation operation and the bit length of variable x, respectively. H_1 and H_2 are two distinct Full Domain Hash (FDH) functions [14], which are defined as $H_1 : \{0,1\}^{|sk|} \rightarrow \{0,1\}^{|p|}$ and $H_2 : \{0,1\}^* \rightarrow \{0,1\}^{|p|}$, respectively, where $sk \overset{R}{\leftarrow} F_p$.

Assumption 1 The cryptographic primitives used in our scheme possess all the required semantic security properties [15]: H_1/H_2 are strong collision-free and secure FDHs [14], producing indistinguishable outputs from the random uniform distribution (i.e., behaves as a Random Oracle [16]). Elliptic Curve Discrete Logarithm Problem (ECDLP) [17] is intractable with appropriate parameters. That is, for a given random point $Q \in E(F_p)$, it is computationally infeasible to determine an integer k such that $Q = kG$, where $G \in \mathbb{G}$.

BAF is an integrated scheme that achieves both forward security and sequential signature aggregation simultaneously. Hence, BAF has a *Key Update* algorithm that follows the "evolve-and-delete strategy" to achieve forward security similar to generic forward secure signatures (e.g., [18]). Moreover, it has *Key Generation*, *Aggregate Signature Generation*,

[1]Delayed detection attack occurs if the log protection mechanism cannot achieve immediate verification (due to the lack of online TTP support). In this case, log verifiers cannot detect whether the log entries are manipulated until the TTP provides necessary keying information to them. Details of this problem is discussed in Section IV.

Aggregate Signature Verification algorithms similar to the aggregate signatures (e.g., [8], [12], [13], [19], [20]).

Definition 1 BAF is four-tuple of algorithms $BAF = (Kg, Upd, ASig, AVer)$ that behave as follows:

- *BAF.Kg:* $BAF.Kg$ is the key generation algorithm, which takes the maximum number of key updates L and identity of signer i (ID_i) as the input and returns L public keys $(pk_0, \ldots, pk_{L-1})$, initial secret key sk_0, and index $n \overset{R}{\leftarrow} F_p$ for ID_i as the output.
- *BAF.Upd:* $BAF.Upd$ is the key update algorithm, which takes the current secret key sk_j where $j \leq L-1$ as the input, and returns the next secret key sk_{j+1} as the output. $BAF.Upd$ also deletes sk_j from the memory.
- *BAF.ASig:* $BAF.ASig$ is the aggregate signature generation algorithm, which takes sk_j, data item $D_j \in \{0,1\}^*$ to be signed, and an aggregate signature $\sigma_{0,j-1}$ (for previously accumulated data items) as the input. It returns an aggregate signature $\sigma_{0,j}$ by folding the individual signature of D_j (i.e., σ_j) into $\sigma_{0,j-1}$.
- *BAF.AVer:* $BAF.AVer$ is the aggregate signature verification algorithm, which takes $(D_0, \ldots, D_j) \in \{0,1\}^*$, its associated aggregate signature $\sigma_{0,j}$, index n and public keys $(pk_0, \ldots, pk_j)$ of ID_i as the input. If the signature is successfully verified, $BAF.AVer$ returns *success*. Otherwise, it returns *failure*. We require that any σ generated by $BAF.ASig$ is accepted by $BAF.AVer$.

Definition 2 BAF defines two *key update models* for the accumulated data: *Per-data item* and *per-time interval* models. In per-data item model, each collected data item is signed and aggregated as soon as it is received. In per-time interval model, in a given time interval t_w, the signer accumulates each collected data item and signs them once at the end of t_w as one large data item. These two models are the same from the perspective of the "evolve-and-delete" strategy; however, they allow a *security-storage trade-off* that can be decided according to application requirements. The per-data item model guarantees forward security of *each individual log entry*, but imposes higher storage overhead on the verifier side. In contrast, the per-time interval model guarantees forward security for *across time intervals*, but incurs less storage overhead. That is, if the attacker compromises the system in t_w, she can forge the log entries accumulated from the beginning of t_w. However, she cannot forge the log entries accumulated before t_w.

BAF behaves according to the *same-signer-distinct-message model* similar to existing forward secure and aggregate logging schemes (e.g, [1], [3], [8], [9]). In this model, the same logger computes aggregate signatures of distinct audit logs accumulated-so-far. This model is an ideal option for secure audit logging applications (e.g., [1]–[3], [5], [9], [21]), since each logger is only responsible for her own audit logs. Note that some aggregate signature schemes (e.g., [12], [13], [19]) use the *different-signer-distinct-message model* (e.g., for secure routing purposes), which is not necessary for secure audit logging.

B. Security Model

The security of BAF is defined as the non-existence of a capable adversary $\mathcal{A}$, confined with certain games, existentially forging a BAF signature even under the exposure of the current keying material. Since BAF aims to achieve both secure sequential signature aggregation and forward security simultaneously, we develop our security model based on the security model of aggregate signatures in [12], [13], [19], [20], generic forward secure signature model in [18], and hybrid security model of FssAgg schemes in [8], [9]. The security model of BAF is defined below:

Definition 3 BAF is an existentially unforgeable forward secure and aggregate signature scheme against adaptive chosen message attacks in the random oracle model [16], if no Probabilistic Polynomial Time (PPT) bounded adversary $\mathcal{A}$ can win the following game with a non-negligible probability.

1) *Setup.* $\mathcal{A}$ is provided with a challenge public key pk_c and parameters L and n.
2) *Queries.* Beginning from $j = 0$, proceeding adaptively, $\mathcal{A}$ is provided with a BAF signing oracle $\mathcal{O}$ under secret key sk_j. For each query, $\mathcal{A}$ supplies a valid BAF signature $\sigma_{0,j-1}$ on some messages $D_0 \in \{0,1\}^*, \ldots, D_{j-1} \in \{0,1\}^*$ signed under $sk_0, \ldots, sk_{j-1}$, where both messages and keys are of her choice. $\mathcal{A}$ also queries an additional message $D_j \in \{0,1\}^*$ of her choice once, which is signed by $\mathcal{O}$ under sk_j. $\mathcal{A}$ then proceeds into the next time period and is provided with oracle $\mathcal{O}$ under sk_{j+1}. The adaptive queries continue, until $\mathcal{A}$ "breaks-in".
3) *Break-in.* When $\mathcal{A}$ decides to break-in in time period t_T, she is allowed to access secret key sk_T.
4) *Forgery.* Eventually, $\mathcal{A}$ halts and outputs an aggregate signature $\sigma^*_{0,t}$ on $D^*_0, \ldots, D^*_t$ under $sk^*_0, \ldots, sk^*_t$. $\mathcal{A}$ wins the game, if (i) $t < T$, (ii) $\sigma^*_{0,t}$ is verified by $BAF.AVer$ successfully, and (iii) $\sigma^*_{0,t}$ is non-trivial (i.e., $\mathcal{A}$ did not ask a signature on D^*_t for time period t in the query phase).

III. The Proposed Scheme

In this section, we present our proposed BAF scheme. We first give an overview of the proposed scheme and then give the detailed description.

A. Overview

The objective of BAF is to achieve six seemingly conflicting goals at the same time to compute and verify forward secure and aggregate signatures for secure audit logging purposes, including high signer computational efficiency, storage efficiency, bandwidth efficiency, public verifiability, immediate verification, and high verifier computational efficiency. To demonstrate how BAF achieves these properties, we first discuss the envisioned design principles, on the basis of which BAF strategy is constructed. We then present the BAF strategy that achieves the stated goals by following these design principles.

Design Principles: BAF is based on the following design principles:

- *Avoid PKC operations for logging:* All existing PKC-based forward secure and aggregate schemes are directly derived from the existing aggregate or forward secure signature schemes. For instance, FssAgg-BLS [8] is based on the aggregate signature scheme given in [19]. Similarly, FssAgg-BM and FssAgg-AR in [1], [3], [9] are based on the forward secure signatures given in [22] and [23], respectively. Hence, existing forward secure and aggregate schemes naturally inherit high computational costs of these signature primitives. To achieve efficiency, BAF restricts operations used in signature generation to basic arithmetic operations and cryptographic hash functions. This implies that BAF does not use any PKC operation for logging.
- *Avoid time factor and online TTP Support:* BAF aims to achieve public verifiability without using any PKC operation at the signer side. One of the possible solutions would be to introduce asymmetry between the signer and the verifiers via the time factor (e.g., TESLA [24]). However, such a scheme cannot achieve immediate verification at the verifier side. Moreover, it requires online TTP support to achieve forward security. (If the signer herself introduces the required asymmetry, then an active attacker compromising the signer can eventually forge the computed signatures). To achieve immediate verification and scalability, BAF uses neither the time factor nor online TTP support.

BAF Strategy: BAF uses a novel strategy called *"Blind-Aggregate-Forward"*. Such a strategy enables signers to log a large number of log entries with little computational, storage, and communication costs in a publicly verifiable way. To achieve this, BAF signature generation has three phases as described below:

1) *Individual Signature Generation:* BAF computes the individual signature of each accumulated data item using a simple and efficient blinding operation. Blinding is applied to the hash of data item via first a multiplication and then an addition operation modular a large prime p by using a pair of secret blinding keys (referred as the blinding key pair). The result of this blinding operation is a unique and random looking output (i.e., the individual signature), which cannot be forged without knowing its associated secret blinding keys.
2) *Key Update:* BAF updates the blinding key pair via two hash operations after each individual signature generation, and then deletes the previous key pair from memory.
3) *Signature Aggregation:* BAF aggregates the individual signature of each accumulated data item into the existing aggregate signature with a single addition operation modular a large prime p, similar to the additive collision-free incremental hash techniques (e.g., [25], [26]).

In the above construction, the individual signature computation binds a given blinding key pair to the hash of signed data item in a specific algebraic form. The signature aggregation maintains this form incrementally and also preserves the indistinguishability of each individual signature. Hence, the resulting aggregate signature can be verified by a set of public key securely. BAF enables this verification by embedding each blinding secret key pair of signer i into a public key pair via an ECC scalar multiplication. Using the corresponding public keys, the verifiers follow the BAF signature verification equation by performing a scalar multiplication for each received data item. The successful verification of the aggregate signature guarantees that only the claimed signer, who possessed the correct blinding secret key pairs before their deletion, could compute such a signature (which is unforgeable after the keys were deleted).

B. Description of BAF

Following the syntax given in Definition 1, the proposed BAF scheme behaves as described below:

1) *BAF.Kg(L, ID_i)*: $BAF.Kg$ generates L private/public key pairs for signer i. The parameter L determines the maximum number of key update operations that a signer can execute, which should be decided according to the application requirements. In BAF, an offline TTP executes $BAF.Kg$ before the initialization of the scheme, and then provides the required keys to system entities.
   a) Pick two random numbers as $(a_0, b_0) \overset{R}{\leftarrow} F_p$, which are *initial blinding keys* of signer i. Also pick a random index number as $n \overset{R}{\leftarrow} F_p$, which is used to preserve the order (sequentiality) of individual signatures. Such an order enforcement is needed, since BAF signature aggregation operation is commutative.
   b) Generate two hash chains from the initial secret blinding keys (a_0, b_0) as $a_{j+1} = H_1(a_j)$ and $b_{j+1} = H_1(b_j)$ for $j = 0, \ldots, L-1$. Also generate a public key for each element of these hash chains as ($A_j = a_j G$ and $B_j = b_j G$) for $j = 0, \ldots, L-1$.
   c) The TTP provides required keys to signer i and verifiers as follows: $ID_i \leftarrow \{a_0, b_0, n\}$ and Verifiers$\leftarrow \{ID_i : A_0, B_0, \ldots, A_{L-1}, B_{L-1}, n\}$.
2) *BAF.Upd(a_l, b_l)*: $BAF.Upd$ is the key update algorithm, which updates the given blinding keys as $a_{l+1} = H_1(a_l)$ and $b_{l+1} = H_1(b_l)$. $BAF.Upd$ then deletes (a_l, b_l) from the memory. $BAF.Upd$ is invoked after each $BAF.ASig$ operation, whose frequency is determined according to the application requirements based on the chosen key update model given in Definition 2 (i.e., per-data item or per-interval key update models).
3) *BAF.ASig($\sigma_{0,l-1}, D_l, a_l, b_l$)*: Assume that signer i has accumulated data items $(D_0, \ldots, D_{l-1})$ and computed the aggregate signature $\sigma_{0,l-1}$. Signer i computes the aggregate signature for new data item D_l via $BAF.ASig$ as follows:
   a) Compute the individual signature σ_l as $\sigma_l = a_l * H_2(D_l || (n + l)) + b_l \bmod p$.
   b) Fold σ_l into $\sigma_{0,l-1}$ as $\sigma_{0,l} = \sigma_{0,l-1} + \sigma_l \bmod p$, where $l > 0$ and $\sigma_{0,0} = \sigma_0$ is a known value.

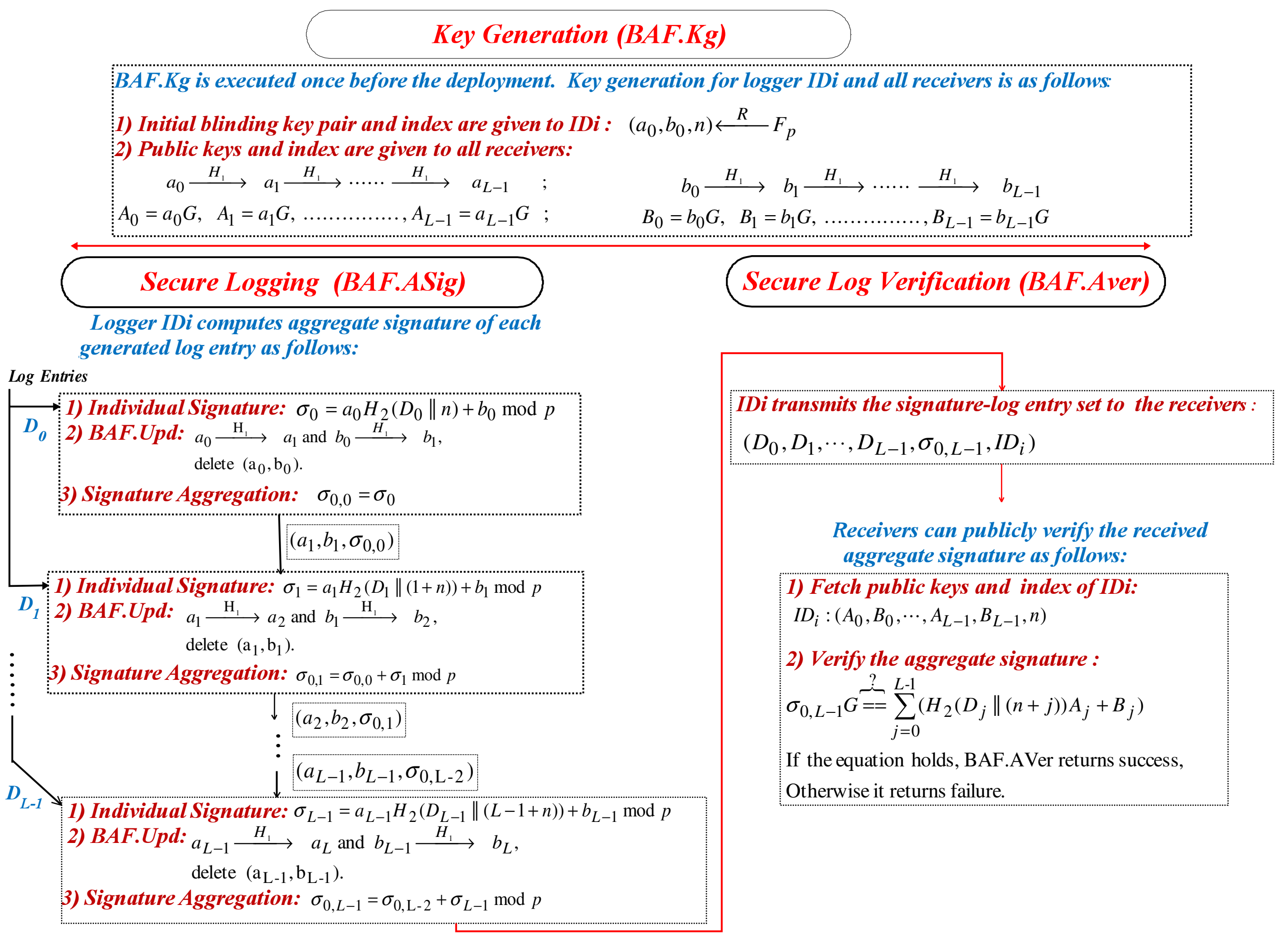
Key Generation (BAF.Kg)
BAF.Kg is executed once before the deployment. Key generation for logger IDi and all receivers is as follows:
1) Initial blinding key pair and index are given to IDi : $(a_0, b_0, n) \xleftarrow{R} F_p$
2) Public keys and index are given to all receivers:
$a_0 \xrightarrow{H_1} a_1 \xrightarrow{H_1} \cdots\cdots \xrightarrow{H_1} a_{L-1}$; $b_0 \xrightarrow{H_1} b_1 \xrightarrow{H_1} \cdots\cdots \xrightarrow{H_1} b_{L-1}$
$A_0 = a_0G,\ A_1 = a_1G, \ldots, A_{L-1} = a_{L-1}G$; $B_0 = b_0G,\ B_1 = b_1G, \ldots, B_{L-1} = b_{L-1}G$
Secure Logging (BAF.ASig)
Logger IDi computes aggregate signature of each generated log entry as follows:
Log Entries
D_0
1) Individual Signature: $\sigma_0 = a_0 H_2(D_0 \| n) + b_0 \bmod p$
2) BAF.Upd: $a_0 \xrightarrow{H_1} a_1$ and $b_0 \xrightarrow{H_1} b_1$, delete (a_0, b_0).
3) Signature Aggregation: $\sigma_{0,0} = \sigma_0$
$(a_1, b_1, \sigma_{0,0})$
D_1
1) Individual Signature: $\sigma_1 = a_1 H_2(D_1 \| (1+n)) + b_1 \bmod p$
2) BAF.Upd: $a_1 \xrightarrow{H_1} a_2$ and $b_1 \xrightarrow{H_1} b_2$, delete (a_1, b_1).
3) Signature Aggregation: $\sigma_{0,1} = \sigma_{0,0} + \sigma_1 \bmod p$
$(a_2, b_2, \sigma_{0,1})$
$(a_{L-1}, b_{L-1}, \sigma_{0,L-2})$
D_{L-1}
1) Individual Signature: $\sigma_{L-1} = a_{L-1} H_2(D_{L-1} \| (L-1+n)) + b_{L-1} \bmod p$
2) BAF.Upd: $a_{L-1} \xrightarrow{H_1} a_L$ and $b_{L-1} \xrightarrow{H_1} b_L$, delete (a_{L-1}, b_{L-1}).
3) Signature Aggregation: $\sigma_{0,L-1} = \sigma_{0,L-2} + \sigma_{L-1} \bmod p$
Secure Log Verification (BAF.Aver)
IDi transmits the signature-log entry set to the receivers :
$(D_0, D_1, \cdots, D_{L-1}, \sigma_{0,L-1}, ID_i)$
Receivers can publicly verify the received aggregate signature as follows:
1) Fetch public keys and index of IDi:
$ID_i : (A_0, B_0, \cdots, A_{L-1}, B_{L-1}, n)$
2) Verify the aggregate signature :
$\sigma_{0,L-1}G \overset{?}{==} \sum_{j=0}^{L-1} (H_2(D_j \| (n+j))A_j + B_j)$
If the equation holds, BAF.AVer returns success,
Otherwise it returns failure.

Fig. 1. BAF algorithms

c) Delete $\sigma_{0,l-1}$ from the memory and invoke $BAF.Upd(a_l, b_l)$.

4) *BAF.AVer*($D_0, \ldots, D_l, \sigma_{0,l}, ID_i$): When the verifier receives data items and their associated aggregate signature from ID_i, she first retrieves public keys (A_j, B_j) for $j = 0, \ldots, l$ and n of ID_i. She then verifies $\sigma_{0,l}$ via the BAF verification equation as follows: $\sigma_{0,l}G \overset{?}{==} \sum_{j=0}^{l}(H_2(D_j||(n+j))A_j + B_j)$. If the equation holds, $BAF.AVer$ returns *success*. Otherwise it returns *failure*.

Figure 1 illustrates the BAF algorithms.

IV. Security Analysis

BAF is proven secure in the following theorem based on the BAF security model given in Definition 3, as long as Assumption 1 holds.

Theorem 1 *Assume there exists a PPT bounded adversary $\mathcal{A}$ that produces an existential forgery against BAF, based on the BAF security model defined in Definition 3 instantiated for L time periods. Assume that $\mathcal{A}$ makes at most q_s signature queries to a BAF signing oracle for this forgery, and eventually succeeds in time τ with probability ϵ. Then we can construct a simulator $\mathcal{B}$ that solves ECDLP by extracting private key (a_c, b_c) from the challenge public key $(A_c = a_cG, B_c = b_cG)$ in time $\tau' = \tau + O(q_s)$ with probability $\epsilon' \geq \epsilon/L$.*

Proof: Assume that forger $\mathcal{A}$ succeeds with probability ϵ in time τ. Then there exists a simulator $\mathcal{B}$ that extracts the target private key (a_c, b_c) from the challenge public key $(A_c = a_cG, B_c = b_cG)$ by using $\mathcal{A}$ as a subroutine, where subscript c denotes the challenge keys. If $\mathcal{A}$ succeeds forging with probability ϵ in time τ, $\mathcal{B}$ succeeds solving ECDLP with a lower bound at ϵ/L within in time $\sim \tau$. We consider the following game, in which $\mathcal{B}$ is given access to a BAF signing oracle $\mathcal{O}$ and interacts with $\mathcal{A}$ as follows:

1) *Setup.* First, simulator $\mathcal{B}$ is given the challenge public key (A_c, B_c) and $n \overset{R}{\leftarrow} F_p$. $\mathcal{B}$ then randomly chooses a *forgery time period* t_w, for which $\mathcal{A}$ is supposed to output her forgery, where $0 \leq w < L$. $\mathcal{B}$ sets public key of t_w to the challenge public key as $(A_w = A_c, B_w = B_c)$ hoping that $\mathcal{A}$ produces a successful forgery for this time period that enables $\mathcal{B}$ to solve the *ECDLP* for (A_c, B_c). $\mathcal{B}$ is also allowed to access $\mathcal{O}$, which signs a given message D under secret key pair (a_c, b_c) (only known by oracle $\mathcal{O}$) and returns σ_c to $\mathcal{B}$. After setting

the challenge public key and forgery time period, $\mathcal{B}$ computes the remaining private/public keys as follows:

a) $\mathcal{B}$ first generates w independent BAF public key pairs as $(A_j = a_j G, B_j = b_j G)$, where each $(a_j, b_j) \overset{R}{\leftarrow} F_p$ for $j = 0, \ldots, w-1$.

b) $\mathcal{B}$ then generates the initial blinding keys for t_{w+1} as $(a_{w+1}, b_{w+1}) \overset{R}{\leftarrow} F_p$, and computes the remaining $(L-w)$ public keys using hash chains $a_{j+1} = H_1(a_j)$ and $b_{j+1} = H_1(b_j)$ as $A_j = a_j G$ and $B_j = b_j G$ for $j = w+1, \ldots, L-1$ (as in $BAF.Kg$ algorithm). $\mathcal{B}$ provides $\mathcal{A}$ with $(A_0, B_0, \ldots, A_{L-1}, B_{L-1}, L, n)$.

Note that even though the secret keys generated by $\mathcal{B}$ are not chained with the target secret keys a_c and b_c through H_1, $\mathcal{A}$ will not be able to distinguish them, since otherwise, $\mathcal{A}$ would be able to distinguish the outputs of H_1 from the random uniform distribution (this implies H_1 is broken).

2) *Queries.* $\mathcal{A}$ can query any BAF signature for any time period t_j of her choice with the condition that after $\mathcal{A}$ queries for t_j, she cannot later query for any $t_{j'} \le t_j$. Beginning from $j = 0$, proceeding adaptively, $\mathcal{A}$ requests a BAF signature from $\mathcal{B}$ *once* [2] on a message D_j of her choice, which will be signed under (a_j, b_j). For each query, $\mathcal{A}$ also supplies $\sigma_{0,j-1}$ on $(D_0, \ldots, D_{j-1})$ under the distinct $(A'_0 = a'_0 G, B'_0 = b'_0 G, \ldots, A'_{j-1} = a'_{j-1} G, B'_{j-1} = b'_{j-1} G)$, where both messages and keys are of her choice. $\mathcal{B}$ then handles query of $\mathcal{A}$ as follows:

a) $\mathcal{B}$ first checks whether $(A'_0, B'_0, \ldots, A'_{j-1}, B'_{j-1})$ are valid, then verifies $\sigma_{0,j-1}$ under these public keys via $BAF.AVer$ algorithm. If any of these controls fail, $\mathcal{B}$ *aborts*. Otherwise, $\mathcal{B}$ continues to the next step.

b) If $j \ne w$, $\mathcal{B}$ computes $\sigma_j = a_j H_2(D_j || (n+j)) + b_j \bmod p$ (since $\mathcal{B}$ knows (a_j, b_j)). Otherwise, $\mathcal{B}$ goes to $\mathcal{O}$ and requests σ_w under (a_w, b_w) (note that only $\mathcal{O}$ knows $(a_w = a_c, b_w = a_c)$).

$\mathcal{B}$ computes $\sigma_{0,j} = \sigma_{0,j-1} + \sigma_j \bmod p$ and returns it to $\mathcal{A}$. $\mathcal{B}$ also maintains a list $\mathcal{L} =< D_j, \sigma_j, \sigma_{0,j} >$ for each computed message-signature pair.

3) *Break-in.* When $\mathcal{A}$ chooses to break-in t_T, she requests secret key of t_T from $\mathcal{B}$. If $t_T \le t_w$, $\mathcal{B}$ *aborts*. Otherwise, $\mathcal{B}$ provides $\mathcal{A}$ with (a_T, b_T).

4) *Forgery.* Eventually, $\mathcal{A}$ halts and outputs a forgery $\sigma^*_{0,t'}$ on $(D^*_0, \ldots, D^*_{t'})$ under distinct public keys $(A^*_0, B^*_0, \ldots, A^*_{t'}, B^*_{t'})$. Forgery of $\mathcal{A}$ is valid if $\sigma^*_{0,t'}$ is non-trivial and valid. That is,

a) $(t' = w) \wedge (D^*_{t'} \ne \mathcal{L}.D_w) \wedge (A^*_{t'} = A_c, B^*_{t'} = B_c)$; and

b) $BAF.AVer(D^*_0, \ldots, D^*_{t'}, \sigma^*_{0,t'}, ID_{\mathcal{A}}) = success$.

If both of the above conditions are satisfied, $\mathcal{B}$ proceeds to solve the *ECDLP* for the challenge public key (A_c, B_c) as follows: $\mathcal{B}$ first isolates σ^*_w from $\sigma^*_{0,w} = \sigma^*_{0,t'}$ as $\sigma^*_w = \sigma^*_{0,w} - \sum_{j=0}^{w-1}(\sigma^*_j) \bmod p$ ($\mathcal{B}$ either knows the required secret key, or maintains the queried σ^*_j in $\mathcal{L}$). Since conditions (a) and (b) are satisfied, $\sigma^*_w G = H_2(D^*_w || (n+w)) A_c + B_c$ holds. $\mathcal{B}$ then fetches $\sigma_w = \sigma_c$ from $\mathcal{L}$ and finds (a_c, b_c) by solving the following modular linear equations:

$$\sigma^*_w = a_c H_2(D^*_w || (n+w)) + b_c \bmod p \quad (1)$$

$$\sigma_c = a_c H_2(D_w || (n+w)) + b_c \bmod p \quad (2)$$

If $(a_c = 0 \vee b_c = 0)$, $\mathcal{B}$ *aborts*. Otherwise, $\mathcal{B}$ returns (a_c, b_c).

In the above game, simulator $\mathcal{B}$ makes as many queries as $\mathcal{A}$ makes. The running time of simulator $\mathcal{B}$ is that of $\mathcal{A}$ plus the overhead due to handling $\mathcal{A}$'s BAF signature queries.

If $\mathcal{A}$ succeeds with probability ϵ in forging, then simulator $\mathcal{B}$ succeeds with probability $\sim (\epsilon/L)$. The argument is summarized as follows: (i) The view of $\mathcal{A}$ that $\mathcal{B}$ produces the signatures is computationally indistinguishable from the view of A interacting with a real BAF signing oracle. That is, if there exists a distinguisher for these two views of $\mathcal{A}$, there exists a distinguisher for H_1. (ii) Conditioned on simulator $\mathcal{B}$ choosing target forgery time period t_w as the period for which $\mathcal{A}$ is supposed to output a valid forgery, the probability that $\mathcal{B}$ solves the ECDLP is the same as the probability that $\mathcal{A}$ succeeds in forgery (i.e., with probability ϵ). Since choosing the "correct" target forgery time period t_w occurs with probability $1/L$, the approximate lower bound on the forging probability of $\mathcal{B}$ is $\sim (\epsilon/L)$. $\square$

Theorem 1 proves that BAF achieves all the required security objectives that a forward secure and aggregate signature scheme must satisfy [1], [8], [9]: Forward security, unforgeability, integrity, authentication, and signature aggregation.

Apart from the above security properties, another security concern in audit logging is *truncation* and *delayed detection* attacks identified in [1], [3]. *Truncation attack* is a special type of deletion attack, in which $\mathcal{A}$ deletes a continuous subset of tail-end log entries. This attack can be prevented via "all-or-nothing" property [8]: $\mathcal{A}$ either should remain previously accumulated data intact, or should not use them at all ($\mathcal{A}$ cannot selectively delete/modify any subset of this data [1]). *Delayed detection attack* targets the audit logging mechanisms requiring online TTP support to enable the log verification. In these mechanisms, the verifiers cannot detect whether the log entries are modified before the TTP provides required keying information. Due to the lack of immediate verification, these mechanisms cannot fulfill the requirement of applications in which the log entries should to be processed in real-time. Ma et al. [1] showed that many existing schemes are vulnerable to these attacks (e.g., [4], [10], [2], [5]).

Based on Theorem 1, it is straightforward to show that BAF is secure against both truncation and delayed detection attacks. The argument is outlined as follows: (i) Theorem 1 guarantees that any data item (or any subset of the accumulated data items), signed and aggregated before the break-in of $\mathcal{A}$, is forward secure and aggregate unforgeable. This implies that BAF achieves "all-or-nothing" property. Thus, it is secure against any attack modifying (or deleting) the data accumulated before the break-in. (ii) In BAF, the verifiers are provided with all the required public keys before deployment. Hence, BAF achieves

[2]Note that $BAF.ASig$ never uses the same blinding key pair to sign two distinct messages, since $BAF.Upd$ immediately updates and then deletes the blind key pair after each signature operation.

TABLE I
NOTATION FOR PERFORMANCE ANALYSIS AND COMPARISON

$Muln$: Modular multiplication mod $n = p'q'$, where p' and q' are large primes		PR: ECC pairing operation
$Mulp$: Modular multiplication mod p	Exp: Modular exponentiation mod p	H: Hash operation
$EMul$: ECC scalar multiplication over F_p	MtP: ECC map-to-point operation	L: max. # of key updates
Sqr: Modular squaring mod n	l: # of data item to be processed	x: # of bits in FssAgg keys
$GSig$: Generic signature generation	$GVer$: Generic signature verification	R: # of verifiers

TABLE II
COMPUTATION INVOLVED IN BAF AND PREVIOUS SCHEMES

	PKC-based					Symmetric
	BAF	FssAgg-BLS [8]	FssAgg-BM [1], [9]	FssAgg-AR [1], [9]	Logcrypt [11]	[2], [4], [5], [8]
Sig	H	$MtP + Exp + Mulp$	$(1 + \frac{x}{2})Muln$	$x \cdot Sqr + (2 + \frac{x}{2})Muln$	$GSig$	H
Upd	$2H$	H	$(x + 1)Sqr$	$(2x)Sqr$	-	H
Ver	$(l + 1) \cdot EMul$	$l \cdot (Mulp + PR)$	$L \cdot Sqr + (l + \frac{l \cdot x}{2})Muln$	$x(L + l)Sqr + 2l(1 + \frac{x}{2})Muln$	$l \cdot GVer$	$l \cdot H$

the immediate verification property, and therefore is secure against delayed detection attack.

V. PERFORMANCE ANALYSIS AND COMPARISON

In this section, we present the performance analysis of our scheme. We also compare BAF with the previous schemes using the following criteria: (i) The computational overhead of signature generation/verification operations; (ii) storage and communication overheads depending on the size of signing key and the size of signature; (iii) scalability properties such as public verifiability and offline/online TTP, and (iv) security properties such as immediate verification and being resilient to the truncation and delayed detection attacks. Computational, storage and communication overheads are critical to justify the practicality of these schemes for task intensive and/or resource-constrained environments. Scalability and security properties are critical to justify the applicability of these schemes in large distributed systems.

We list the notation used in our performance analysis and comparison in Table I. Based on this notation, for each of the above category, we first provide the analysis of BAF, and then present its comparison with the previous schemes both analytically and numerically. Note that we accept the per-data item key update model as the comparison basis in our analysis.

A. Computational Overhead

We first analyze the computational overhead of BAF for signing a single log entry. Individual signature generation requires one H, one addition and one multiplication modular p. Key update requires $2H$, and the signature aggregation requires one addition modular p. Since the overhead of addition and multiplication operations is negligible, the total cost of signing a single log entry is only $3H$.

We now analyze the signature verification overhead of BAF. By following the BAF signature verification equation, verifying a single log entry requires one $EMul$, one H and one ECC addition. Note that it is possible to avoid the ECC addition by using an optimization: In the key generation phase, we can compute and release $B'_j = \sum_{i=0}^{j} B_j = (\sum_{i=0}^{j} b_j)G$ instead of $B_j = b_j G$ for $j = 0, \ldots, L-1$ to speed up the signature verification. In this way, the verifiers can perform the signature verification with only one ECC addition (negligible cost) regardless of the value of l as $\sigma_{0,l-1} * G \overset{?}{==} \sum_{j=0}^{l-1}(H_2(D_j||(n+j)) * A_j) + B'_{l-1}$. The cost of H is negligible in this case, since the total computational cost is dominated by $EMul$. Hence, the aggregate signature verification cost of BAF for l received log entries is $(l+1) \cdot EMul$.

To get an intuitive feeling about the computational overhead, we measured the execution times of basic BAF operations on a laptop with a 1.60GHz Pentium D processor and 512MB RAM running Windows XP. We used MIRACL library [27] compiled with Visual C++ 2005 for necessary cryptographic operations. A single $EMul$ operation over 160 bit random EC takes 2.05 ms, while a single H operation (i.e, SHA-1) takes 0.02 ms. Hence, the execution times of signing and verifying a single log entry for BAF can be estimated as 0.06 ms and (2.05+0.06)=2.11 ms, respectively.

Comparison: The closest counter parts of our scheme are FssAgg schemes [1], [3], [8], [9]. The signature generation of FssAgg-BLS [8] is expensive due to Exp and MtP, while its signature verification is highly expensive due to pairing operations. Different from FssAgg-BLS, FssAgg-BM and FssAgg-AR [9] rely on efficient PKC operations such as Sqr and $Muln$. However, these schemes are also computationally costly, since they require heavy use of such PKC operations. For instance, FssAgg-BM [9] requires $(x+1)Sqr + (1 + x/2)Muln$ (i.e., x≅160 [9]) for the signature generation (key update plus the signing cost), and it requires $L \cdot Sqr + (l + x \cdot l/2)Muln$ for the signature verification. Similarly, FssAgg-AR requires $(3x)Sqr + (2 + x/2)Muln$ for the signature generation, and it requires $x(L+l)Sqr + 2l(1 + \frac{x}{2})Muln$ for the signature verification. Logcrypt uses a digital signature scheme (e.g., ECDSA) to sign and verify each log entry separately without signature aggregation [11], and thus has standard signature costs. The symmetric schemes [2], [4], [5], [8] are in general efficient, since they only need symmetric cryptographic operations. Table II summarizes the computational costs of all the compared schemes.

Table III shows the estimated execution time of BAF and previous schemes. The execution times of BAF, Logcrypt, and the symmetric schemes [2], [4], [5], [8] were obtained using the MIRACL library [27] on a laptop with a 1.60GHz Pentium D processor and 512MB RAM, while the results for all FssAgg

TABLE III
ESTIMATED EXECUTION TIME (IN MS) OF BAF AND PREVIOUS SCHEMES FOR A SINGLE LOG ENTRY

	PKC-based					Symmetric
	BAF	FssAgg-BLS [8]	FssAgg-BM [1], [9]	FssAgg-AR [1], [9]	Logcrypt [11]	[2], [4], [5], [8]
Sig	0.06	30.0	5.55	11.66	2.11	0.06
Ver	2.11	33.0	2.2	8.1	12.09	0.06

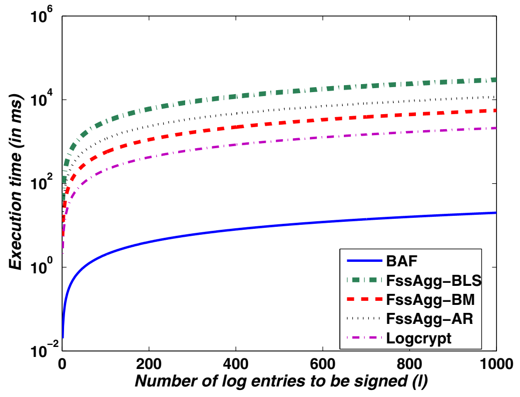

Fig. 2. Signing time comparison (in ms)

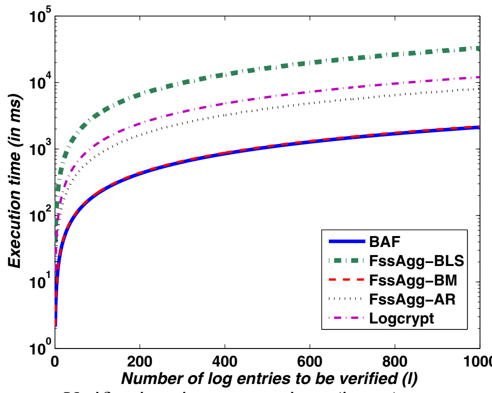

Fig. 3. Verification time comparison (in ms)

schemes are taken from [1], [9], which used a laptop with a 1.73GHz Intel dual-core with 1GB RAM. Thus, our numerical comparison of BAF and the FssAgg schemes is conservative for BAF.

When compared with PKC-based FssAgg-BLS, FssAgg-BM, FssAgg-AR and Logcrypt, BAF is 500, 92, 194, and 35 times faster for loggers, respectively. This efficiency makes BAF the best alternative among all existing schemes for secure logging with public verifiability in task intensive and/or resource-constrained applications. Similarly, BAF signature verification is also more efficient than the previous schemes. When compared with FssAgg-BLS, FssAgg-AR and Logcrypt, BAF is 15.6, 3.8 and 5.7 times faster, respectively. BAF is also slightly more efficient than FssAgg-BM. Figure 2 and Figure 3 further show the comparison of BAF and the previous schemes that allow public verification in terms of signature generation and verification time as the number of log entries increases. These figures clearly show that BAF is the most computationally efficient one among all these choices.

When compared with the previous symmetric logging schemes (e.g., [2], [4], [5], [8], [10]), BAF signature generation is equally efficient even though it is a PKC-based scheme. However, signature verification of the symmetric logging schemes is more efficient than all the existing PKC-based schemes, including BAF. Note that these symmetric schemes sacrifice storage/communication efficiency, public verifiability, and certain security properties (e.g., truncation and delayed detection attacks) to achieve this verifier efficiency. Overall, the advantages of BAF over the symmetric logging schemes include its public verifiability, high storage and communication efficiency, and scalability.

B. Storage and Communication Overheads

In BAF, the size of signing key is $2|p|$ (e.g., $|p|$=512 bit), and the size of authentication tag is $|p|$. Since BAF allows signature aggregation, independent from the number of data items to be signed, the size of resulting authentication tag is always constant, which is equal to $|p|$. Furthermore, BAF derives the current signing key from the previous one, and then deletes the previous signing key from the memory. Hence, the size of signing key is also constant, which is equal to $2|p|$. Based on these parameters, both the storage and communication overheads of BAF are small and constant (i.e., $3|p|$ and $|p|$, respectively).

Comparison: We use the storage and communication overheads of loggers as the comparison basis. The storage and communication overheads are measured according to the size of a single signing key, the size of a single authentication tag, and the growth rate of these two parameters with respect to the number of data items to be processed, that is, whether they grow linear, or remain constant for the increasing number of data items to be processed. Table IV summarizes the comparison.

Bellare-Yee scheme I [10] and scheme II [4] (denoted BY I and BY II, respectively), Schneier-Kelsey scheme I [5] and scheme II [2] (denoted SK I and SK II, respectively), and FssAgg-MAC [8] all use a MAC function to compute an authentication tag for each log entry with a different key, where the sizes of key and resulting tag are both $|H|$ (e.g., 160 bit). Logcrypt [11] extended the idea given in [4], [5] by replacing MAC with a digital signature such as ECDSA, where the size of signing key is $|q|$ (e.g., 160 bit) and the size of resulting signature is $2|q|$, respectively. All these schemes incur high storage and communication overheads to the logger. They cannot achieve signature aggregation, and therefore they require storing/transmitting an authentication tag for each log entry. Hence, the storage and communication overheads of these symmetric schemes [2], [4], [5], [10] and Logcrypt [11] are all linear as $O(L) * |H|$ and $O(L) * |q|$, respectively. Different from these schemes, FssAgg-MAC achieves sig-

TABLE IV
STORAGE AND COMMUNICATION OVERHEADS OF BAF AND PREVIOUS SCHEMES

Criteria	BAF	FssAgg Schemes [1], [3], [8], [9]				Logcrypt [11]	BY I [10], BY II [4]	SK I [5], SK II [2]
		BLS	BM	AR	MAC			
Key Size	$2\|p\|$	$\|q\|$	$(x+1)\|n\|$	$2\|n\|$	$\|H\|$	$\|q\|$	$\|H\|$	$\|H\|$
Signature Size	$\|p\|$	$\|p\|$	$\|n\|$	$\|n\|$	$\|H\|$	$2\|q\|$	$\|H\|$	$\|H\|$
Storage Cost	$3\|p\|$	$\|p\|+\|q\|$	$(x+2)\|n\|$	$3\|n\|$	$O(R)*\|H\|$	$O(L)*\|q\|$	$O(L)*\|H\|$	$O(L)*\|H\|$
Comm. Cost	$\|p\|$	$\|p\|$	$\|n\|$	$\|n\|$	$\|H\|$	$O(L)*2\|q\|$	$O(L)*\|H\|$	$O(L)*\|H\|$

TABLE V
SCALABILITY AND SECURITY PROPERTIES OF BAF AND PREVIOUS SCHEMES

Criteria	BAF	FssAgg Schemes [1], [3], [8], [9]				Logcrypt [11]	BY I [10], BY II [4]	SK I [5], SK II [2]
		BLS	BM	AR	MAC			
Public Verifiability	Y	Y	Y	Y	N	Y	N	N
Offline TTP	Y	Y	Y	Y	Y	Y	N	N
Immediate Verification	Y	Y	Y	Y	Y	Y	N	N
Resilient to Delayed Detection Attack	Y	Y	Y	Y	Y	Y	N	N
Resilient to Truncation (Deletion) Attack	Y	Y	Y	Y	Y	N	N	N

nature aggregation, and its communication overhead is only $|H|$. However, since FssAgg-MAC requires symmetric key distribution, its storage overhead is also linear (i.e., $O(R)|H|$).

The PKC-based FssAgg-BLS [8], FssAgg-BM and FssAgg-AR [9] achieve signature aggregation in a publicly verifiable way, and therefore their storage and communication overheads are constant. Table IV shows that they are efficient in terms of both the storage and communication overheads, with the exception of FssAgg-BM, which is slightly more costly (i.e., $(x+2)|n|$).

BAF has constant and small storage and communication overheads, and is significantly more efficient than all the schemes that incur linear storage and communication overheads (e.g., [2], [4], [5], [8], [10], [11]). BAF is also more efficient than FssAgg-AR/BM [9] and as efficient as FssAgg-BLS [8], as shown in Table IV.

C. Scalability and Security

BAF can produce forward secure and aggregate signatures that are publicly verifiable via the signers' corresponding public key sets. Also, BAF does not need online TTP support for the signature verification, since the verifiers can store all the required keying material without facing a key exposure risk. (Note that in the symmetric schemes such as [2], [4], [5], [10], the verifiers cannot store the verification keys on their own memory, since $\mathcal{A}$ compromising a verifier can obtain all the secret keys of all signers.) Furthermore, BAF does not use the time factor to be publicly verifiable, and therefore achieves immediate verification. Finally, BAF is proven to be secure against the truncation and delayed detection attacks (see Section IV). Hence, BAF achieves all the desirable scalability and security properties simultaneously.

Comparison: Table V shows the comparison of BAF with the previous schemes in terms of their scalability and security properties. The symmetric schemes BY I, BY II [4], [10], SK I, and SK II [2], [5] cannot achieve public verifiability. Moreover, they require online TTP support to enable log verification. The lack of public verifiability and the requirement for online TTP significantly limit their applicability to large distributed systems. Furthermore, they are vulnerable to both truncation and delayed detection attacks [1], [3]. FssAgg-MAC [8] does not need online TTP and is secure against the aforementioned attacks. However, FssAgg-MAC is also a symmetric scheme, which is not publicly verifiable. Hence, none of the previous symmetric schemes can fulfill the requirements of large distributed systems.

PKC-based FssAgg schemes [1], [3], [8], [9] and Logcrypt [11] are publicly verifiable. They do not need online TTP support, and can achieve immediate verification. These schemes are also secure against the truncation and delayed detection attacks, with the exception of Logcrypt in [11].

BAF, achieving all the required scalability and security properties, is also much more computational and storage efficient than FssAgg schemes [1], [3], [8], [9]. Hence, BAF is the most efficient scheme among the existing alternatives that can achieve all the desirable secure auditing properties simultaneously.

VI. RELATED WORK

The pioneering studies addressing the forward secure stream integrity for audit logging were presented in [4], [10]. The main focus of these schemes is to formally define and analyze forward-secure MACs and PRNGs. Based on their forward-secure MAC construction, they also presented a secure logging scheme, in which log entries are tagged and indexed according to the evolving time periods. Schneier et al. [2], [5] proposed secure logging schemes that use one-way hash chains together with forward-secure MACs to avoid using tags and indexes. Logcrypt [11] extended the idea given in [4], [5] to PKC domain by replacing MACs with digital signatures and ID-based cryptography. Finally, Ma et al. proposed a set of comprehensive secure audit logging schemes in [1], [3] based on their forward secure and aggregate signature schemes given in [8], [9]. The detailed analysis and comparison of all these schemes with ours were given in Section V.

Apart from the above schemes, Chong et al. extended the scheme in [5] by strengthening it via tamper-resistant hardware [28]. Moreover, Waters et al. proposed an audit log scheme that enables encrypted search on audit logs via Identity-Based Encryption (IBE) [21]. These works are complementary to ours.

VII. Conclusion

In this paper, we developed a new forward secure and aggregate audit logging scheme for large distributed systems, which we refer to as *Blind-Aggregate-Forward (BAF)* logging scheme. BAF simultaneously achieves six seemingly conflicting goals for secure audit logging, including very low logger computational overhead, near-zero storage and communication overheads, public verifiability (without online TTP support), immediate log verification, and high verifier efficiency. Our comparison with the previous alternative approaches demonstrate that BAF is the best choice for secure audit logging in large distributed systems, even for task intensive and/or resource constrained environments.

In our future work, we will investigate the integration of BAF in distributed systems such as virtual computing clouds. We would like to examine and identify system level issues involved in secure audit logging on untrusted platforms.

Acknowledgements

This work is supported by the U.S. National Science Foundation under grant CAREER-0447761 and the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government. The material contained in this paper has been cleared through all the authors affiliations.

References

[1] D. Ma and G. Tsudik, “A new approach to secure logging,” *ACM Transaction on Storage (TOS)*, vol. 5, no. 1, pp. 1–21, 2009.

[2] B. Schneier and J. Kelsey, “Secure audit logs to support computer forensics,” *ACM Transaction on Information System Security*, vol. 2, no. 2, pp. 159–176, 1999.

[3] D. Ma and G. Tsudik, “A new approach to secure logging,” 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security, DBSEC ’08, London, UK., 2008.

[4] M. Bellare and B. S. Yee, “Forward-security in private-key cryptography,” in *CT-RSA*, 2003, pp. 1–18.

[5] B. Schneier and J. Kelsey, “Cryptographic support for secure logs on untrusted machines,” in *Proceedings of the 7th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 1998.

[6] K. Fall, “A delay-tolerant network architecture for challenged internets,” in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, SIGCOMM ’03*. New York, NY, USA: ACM, 2003, pp. 27–34.

[7] D. Halperin, T. Kohno, T. Heydt-Benjamin, K. Fu, and W. Maisel, “Security and privacy for implantable medical devices,” *Pervasive Computing, IEEE*, vol. 7, no. 1, pp. 30–39, Jan.-March 2008.

[8] D. Ma and G. Tsudik, “Forward-secure sequential aggregate authentication,” in *Security and Privacy, SP ’07. IEEE Symposium on Security and Privacy*, 20-23 May 2007, pp. 86–91.

[9] D. Ma, “Practical forward secure sequential aggregate signatures,” in *ASIACCS ’08: Proc. of the 2008 ACM symposium on Information, computer and communications security*. NY, USA: ACM, 2008, pp. 341–352.

[10] M. Bellare and B. S. Yee, “Forward integrity for secure audit logs,” ftp://www.cs.ucsd.edu/pub/bsq/pub/fi.ps, 1997.

[11] J. E. Holt, “Logcrypt: forward security and public verification for secure audit logs,” in *ACSW Frontiers ’06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*. Australia: Australian Computer Society, Inc., 2006, pp. 203–211.

[12] A. Boldyreva, C. Gentry, A. O’Neill, and D. Yum, “Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing,” in *Proceedings of the 14th ACM conference on Computer and communications security, CCS ’07*. New York, NY, USA: ACM, 2007, pp. 276–285.

[13] Y. Mu, W. Susilo, and H. Zhu, “Compact sequential aggregate signatures,” in *SAC ’07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 249–253.

[14] M. Bellare and P. Rogaway, “The exact security of digital signatures: How to sign with rsa and rabin.” Springer-Verlag, 1996, pp. 399–416.

[15] O. Goldreich, *Foundations of Cryptography*. Cambridge University Press, 2001, vol. Basic Tools.

[16] M. Bellare and P. Rogaway, “Random oracles are practical: a paradigm for designing efficient protocols,” in *CCS ’93: Proceedings of the 1st ACM conference on Computer and communications security*. NY, USA: ACM, 1993, pp. 62–73.

[17] D. Boneh, “The decision diffie-hellman problem,” in *Proceedings of the Third Algorithmic Number Theory Symposium, LNCS*, 1998, pp. 48–63.

[18] H. Krawczyk, “Simple forward-secure signatures from any signature scheme,” in *Proceedings of the 7th ACM conference on Computer and Communications Security, CCS ’00*. NY, USA: ACM, pp. 108–115.

[19] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, “Aggregate and verifiably encrypted signatures from bilinear maps,” in *Proceedings of Advances in Cryptology (EUROCRPYT 2003)*. Springer, 2004, pp. 416–432.

[20] A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham, “Sequential aggregate signatures from trapdoor permutations,” *EUROCRYPT*, 2004.

[21] B. Waters, D., G. Durfee, and D. Smetters, “Building an encrypted and searchable audit log,” in *ACM Annual Symposium on Network and Distributed System Security*, 2004.

[22] M. Bellare and S. Miner, “A forward-secure digital signature scheme,” in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO ’99*. Springer-Verlag, pp. 431–448.

[23] M. Abdalla and L. Reyzin, “A new forward-secure digital signature scheme,” in *ASIACRYPT ’00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*. London, UK: Springer-Verlag, 2000, pp. 116–129.

[24] A. Perrig, R. Canetti, D. Song, and D. Tygar, “Efficient authentication and signing of multicast streams over lossy channels,” in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[25] M. Bellare and D. Micciancio, “A new paradigm for collision-free hashing: incrementality at reduced cost,” in *In Eurocrypt97*. Springer-Verlag, 1997, pp. 163–192.

[26] B.-M. Goi, M. Siddiqi, and H.-T. Chuah, “Computational complexity and implementation aspects of the incremental hash function,” *IEEE Transactions on Consumer Electronics*, vol. 49, no. 4, 2003.

[27] Shamus Software, “Multiprecision integer and rational arithmetic c/c++ library (MIRACL),” http://www.shamus.ie/.

[28] C. N. Chong and Z. Peng, “Secure audit logging with tamper-resistant hardware,” in *18th IFIP International Information Security Conference (IFIPSEC)*. Kluwer Academic Publishers, 2003, pp. 73–84.