A New Approach to Secure Logging

Di Ma and Gene Tsudik

Computer Science Department,
University of California, Irvine
{dma1,gts}@ics.uci.edu

Abstract. The need for secure logging is well-understood by the security professionals, including both researchers and practitioners. The ability to efficiently verify all (or some) log entries is important to any application employing secure logging techniques. In this paper, we begin by examining state-of-the-art in secure logging and identify some problems inherent to systems based on trusted third-party servers. We then propose a different approach to secure logging based upon recently developed Forward-Secure Sequential Aggregate (FssAgg) authentication techniques. Our approach offers both space-efficiency and provable security. We illustrate two concrete schemes – one private-verifiable and one public-verifiable – that offer practical secure logging without any reliance on on-line trusted third parties or secure hardware. We also investigate the concept of immutability in the context of forward secure sequential aggregate authentication to provide finer grained verification. Finally, we report on some experience with a prototype built upon a popular code version control system.
KEYWORDS: secure logging, MACs, signatures, forward secure stream integrity, truncation attack

1 Introduction

System logs are an important part of any secure IT system. They record noteworthy events happened in the past such as user activity, program execution status, system resource usage, data changes, etc. They provide a valuable view of the past and current states of almost any type of complex system. In conjunction with appropriate tools and procedures, audit logs can be used to enforce individual accountability, reconstruct events, detect intrusions and identify problems. Keeping system audit trails and reviewing them in a consistent manner is recommended by NIST as one of the good principles and practices for securing computer systems [1]. Most modern software servers include some sort of logging mechanisms.

Because of their forensic value, system logs are an obvious target for attackers. An attacker who gains access to a system naturally wishes to remove traces of her presence in order to hide attack details or frame innocent users. In fact, the first target of an experienced attacker would often be the logging system [2, 3]. To make the audit log secure, we must prevent the attacker from modifying log data. Secure versions of audit logs should be designed to defend against such tampering. Providing *integrity* checks, the primary security requirement for any secure logging system, is informally stated in the Orange book [4] as:

Audit data must be protected from modification and unauthorized destruction to permit detection and after-the-fact investigation of security violations.

In addition to the traditional meaning of *data integrity* which stipulates no insertion of fake data and no modification or deletion of existing data, the integrity of a log file requires, in addition, no re-ordering of log entries. That is, we require *stream integrity* for securing audit logs.

In many real-world applications, a log file is generated and stored on an untrusted logging machine which is not physically secure enough to guarantee that it cannot be compromised [5]. Compromise of a logging machine can happen as long as the Trusted Computing Base (TCB) – the system component responsible for logging – is not totally bug-free, which is unfortunately always the case. In systems using *remote logging* (which send audit data to a remote trusted server), if the server is not available, the log has to be buffered and stored temporarily at the local machine. Once an attacker obtains the current secret key of the compromised logging machine, she can modify *post-compromise* data at will. In this case, one important issue is *forward integrity*: how to ensure that *pre-compromise* data cannot be manipulated. That is, even if the attacker obtains the current secret key, she must be unable to modify audit data generated before compromise.

No security measure can protect log entries created after an attacker gains control of a logging machine, unless keys are periodically updated with the help of a remote trusted server or a local trusted hardware component (e.g., using key-insulated and intrusion-resilient authentication schemes [6–8]). We focus on the security of log entries made before the compromise of a logging machine. Consequently, we require *forward-secure stream integrity*, i.e., resistance against post-compromise insertion, alteration, deletion and re-ordering of pre-compromise log entries.

Traditional log integrity techniques include using special write-only hard disks or remote logging where copies of log entries are sent to several geographically different remote machines. Recently a number of cryptographic approaches have been proposed to address log integrity security that are generated and stored on the local logging server [2, 3, 5, 9]. Bellare and Yee were the first to define the *forward-secure stream integrity* property required in an audit log system and proposed to use forward-secure MACs and index log entries [2, 3]. Schneier and Kelsey proposed a similar system based on forward-secure MACs and one-way hash chain [5]. Holt extended Schneier and Yee's system to the public key setting [9]. Unfortunately, all these schemes do not defend against *truncation attack* - a special kind of deletion attack whereby the attacker deletes a contiguous subset of tail-end log entries. Furthermore, private key based schemes like Schneier-Kelsey and Bellare-Yee also suffers from *delayed detection attack* [1] since they needs a trusted server to aid users in verifying the integrity of the audit log; modifications can not be detected until the entire log data is uploaded to the trusted server. Also all prior schemes are inefficient in terms of both storage and

[1] For a precise definition, refer to Section 2.

communication costs which make them impractical to be used on devices with commensurately meager resources such as an implantable medical device [10]. We present a detailed analysis of prior schemes in Section 2.

In order to mitigate aforementioned bad features of prior schemes, we propose a new scheme which provides *forward-secure stream integrity* for audit logs generated and stored on untrusted logging machines. Our scheme is based on a new cryptographic technique called *forward-secure aggregate* (FssAgg) authentication recently proposed in [11, 12]. In a FssAgg authentication scheme, forward-secure signatures (or MACs) generated by the same signer are combined sequentially into a single aggregate signature. Successful verification of an aggregate signature is equivalent to that of each component signature. (However, as discussed later, failed verification of an aggregate signature only implies that at least one component signature is invalid.) Thus, a secure FssAgg signature scheme is a good match for secure logging applications – it resists truncation attacks due to its all-or-nothing (aggregate and forward-secure) signature verification. In our scheme, users themselves can verify the log – without relying on a trusted server – which obviates delayed detection attacks. Our scheme offers storage and bandwidth efficiency inherited from the underlying FssAgg scheme. Also, depending on the specific FssAgg scheme used, our scheme can be either private- or public-verifiable.

In a FssAgg scheme, individual signatures are erased once they are folded into the aggregate signature. Subsequent validity of individual log entries is implied by the validity of the aggregated signature computed over all log entries. This indirect verification process is costly if the verifier is only interested in the validity of one specific log entry. The need to provide finer-grained verification in certain applications motivates us to keep individual log entry signatures in the log file. However since the aggregation function is public, revealing individual signatures enables anyone to truncate log entries and create new aggregate signature based on existing ones. To prevent this truncation attack (even when individual component signatures are revealed) we extend the FssAgg authentication scheme to be immutable. (Informally, immutability means the difficulty of computing new valid aggregated signatures from existing signatures.)

1.1 Contributions

Our contributions are as follows:

1. We identify some fundamental security issues and architectural limitations in prior secure logging schemes.
2. We propose new secure logging schemes based on recently proposed FssAgg authentication techniques. Our schemes provide *forward-secure stream integrity* for audit logs generated and stored on untrusted logging machines and avoid the undesirable features of prior schemes. Our schemes inherit the provable security of the underlying FssAgg schemes.
3. We study immutability in the context of FssAgg authentication and extend existing FssAgg MAC/signature schemes to support immutability and provide finer-grained verification.
4. We evaluate proposed schemes by comparing them with previous schemes, in terms of security as well as communication and computation efficiency. Our evaluation

shows that our schemes offer better security and incur less computation and communication overhead.
5. We implement existing FssAgg signature schemes and compare their performance in the context of secure logging systems.

Organization: We begin with the analysis of the state-of-the-art in Section 2, followed by introduction of forward-secure aggregate authentication in Section 3. We then show how to use FssAgg schemes in logging applications: we propose a private-verifiable scheme in Section 4 and a public-verifiable scheme in Section 5. Next, we present immutable FssAgg schemes in 6. We evaluate our schemes in Section 7 and report on some experience with a prototype implementation in Section 8. Section 9 overviews related work and Section 10 concludes the paper.

2 Current Approach Analysis

In this section, we examine the state-of-the-art represented by Schneier-Kelsey scheme [5]. It has been used as a foundation by many subsequently proposed secure logging systems. Readers interested in further details of the Schneier-Kelsey scheme are referred to [5].

2.1 Overview of Schneier-Kelsey Scheme

In the Schneier-Kelsey scheme, a logging machine $\mathcal{U}$ opening a new audit log first establishes a shared secret key A_0 with a trusted remote server $\mathcal{T}$. After each audit entry is generated, the current secret key A_i is evolved into A_{i+1} through a one-way function. Log entries are linked using a hash chain. Each log entry L_i contains three parts:

1. log entry data[2] M_i;
2. element Y_i in the hash chain where[3]:

$$Y_i = H(M_i||Y_{i-1}) \ \textbf{ and } \ Y_0 = H(M_0)$$

3. forward-secure MAC denoted as Z_i, computed over Y_i with the current secret key: $Z_i = MAC_{A_i}(Y_i)$.

$\mathcal{U}$ closes the log file by writing a special final-record message, D_f and erasing A_f as well as other secrets, if any.

There is no constant high-bandwidth channel between $\mathcal{U}$ and $\mathcal{T}$. It is assumed that $\mathcal{U}$ communicates log entries infrequently to $\mathcal{T}$. At times, a moderately-trusted person or machine, called $\mathcal{V}$, may need to verify or read the audit log, while it is still on $\mathcal{U}$. $\mathcal{V}$

[2] [5] also provides access control to audit log. Each log entry L_i contains a log entry type W_i and $C_i = E_{K_i}(D_i)$: the actual log data D_i encrypted under an access control key K_i. As we only focus on integrity of audit log, in this paper, we replace the two elements W_i and C_i as M_i to make our explanation clearer.

[3] In the Schneier-Kelsey scheme, Y_i is calculated as $Y_i = H(W_i||C_i||Y_{i-1})$

receives from $\mathcal{U}$ a copy of the audit log, $[L_0, L_1, \cdots, L_f]$, where f is the index value of the last record, from $\mathcal{U}$. $\mathcal{V}$ goes through the hash chain in the log entries (the Y_i values), verifying that each entry in the hash chain is correct. $\mathcal{V}$ then sends Y_f and Z_f to $\mathcal{T}$. $\mathcal{T}$ knows A_0 so he can calculate A_f; this allows him to verify that $Z_f = MAC_{A_f}(Y_f)$. $\mathcal{T}$ informs $\mathcal{V}$ about the verification result and $\mathcal{V}$ discovers whether the received copy of the log has any problems.

2.2 Analysis

We observe that the Schneier-Kelsey scheme has two security-related drawbacks:

Truncation Attack: a kind of deletion attack whereby the attacker erases a contiguous subset of tail-end log messages. The truncation attack represents a real danger and is valuable to an attacker. After breaking in, a natural goal for an attacker is to modify the audit log by deleting the most recent log entries generated right before the break-in.

The Schneier-Kelsey scheme uses a hash chain to link log entries such that that undetectable log (link) deletion is impossible. This pertains to log entries already farmed out to $\mathcal{T}$. However, log entries still residing on $\mathcal{U}$ are vulnerable to the truncation attack since there is no single authentication tag protecting the integrity of the entire log file. A hash chain element Y_i only protects data records generated before time i. Thus, truncating log entries generated after time i will not be detected by $\mathcal{T}$, unless there is a synchronization between $\mathcal{U}$ and $\mathcal{T}$ and the latter knows the current value of f. Without a continuous communication channel, synchronization between $\mathcal{U}$ and $\mathcal{T}$ would require $\mathcal{U}$ to generate log entries at a fixed rate. But, most logging systems are event-driven and events are unevenly spaced. Logging events at a fixed rate hinders the logging machine's ability to fully utilize its computation and storage resources.

Delayed Detection: Recall that, in the Schneier-Kelsey scheme, $\mathcal{V}$ is unable to verify a log file by itself and needs to ask for help from $\mathcal{T}$. If this occurs before $\mathcal{T}$ has received a copy of the most up-to-date log from $\mathcal{U}$, and before $\mathcal{U}$ has closed the log file, an attacker can modify pre-compromise records without being detected. Albeit, such modification will eventually be detected later, after $\mathcal{T}$ receives the updated version of a log file.

We illustrate the delayed detection attack in Figure 1. Suppose that, at time a (≥ 0), $\mathcal{U}$ has transmitted log entries $[L_0, \cdots, L_a]$ to $\mathcal{T}$. At time b $(> a)$, an attacker breaks into $\mathcal{U}$ and obtains the current secret key A_b. Even though the attacker can not recover secret keys used in time intervals $[a{+}1, b{-}1]$, she can modify the values of M_i and their corresponding Y_i in this interval without touching the values of Z_i. At time f $(\geq b)$, $\mathcal{V}$ receives a copy of log entries $L_0, \cdots, L_f$. $\mathcal{V}$ sends Y_f and Z_f to $\mathcal{T}$. Since the attacker knows A_b at break-in, she can generate valid MACs from time b. The verification of Y_f with Z_f at $\mathcal{T}$ will succeed. The modified log file will translate false information to $\mathcal{V}$ and activities conducted within interval $[a + 1, f]$ will elude $\mathcal{V}$'s detection. In Figure 1, values in the shaded area (M and Y values in time interval $[a{+}1, b{-}1]$, all values within $[b, f]$) can be manipulated by an attacker. Since there is no continuous high-bandwidth $\mathcal{U} \leftrightarrow \mathcal{T}$ communication channel and $\mathcal{U}$ only communicates with $\mathcal{T}$ infrequently, the time interval $[a + 1, f]$ can be potentially large.

Since the attacker is unable to fake any values Z_i (for $i \in [a+1, b-1]$), any manipulation in this period can be detected whenever the corresponding log entries are uploaded to $\mathcal{T}$ and $\mathcal{T}$ scan-verifies all individual MACs.[4]

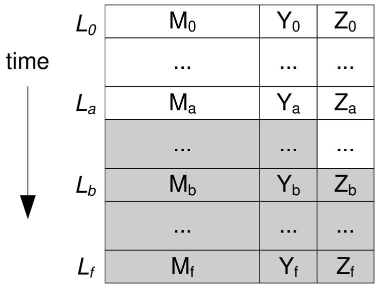

Fig. 1. Log file under delayed detection attack. Data in shaded area can be controlled by an attacker. a: time at which log entries are uploaded to $\mathcal{T}$; b: break-in time; f: index value of last log entry as well as time when $\mathcal{V}$ receives a copy of log file from $\mathcal{U}$.

The two drawbacks of the Schneier-Kelsey scheme seem to be quite fundamental. However, it is rather surprising that they have not been addressed by any later work. In addition to the security issues discussed above, the Schneier-Kelsey scheme has some architectural limitations:

Online Server. As mentioned earlier, the Schneier-Kelsey scheme employs a server-assisted verification process and a trusted server $\mathcal{T}$ must be present whenever $\mathcal{V}$ wants to pose an integrity query. In other words, the scheme requires a continuous channel (not necessarily high-width in case of the integrity check) between $\mathcal{V}$ and $\mathcal{T}$. As with any centralized solution, the Schneier-Kelsey scheme has the single point of failure. Furthermore, the overall security of the Schneier-Kelsey scheme relies on the frequency of communication between $\mathcal{U}$ and $\mathcal{T}$. Basically, the need for the on-line server can be avoided by using a public key based approach as in [9].

Storage Inefficiency. Each log entry contains a hash value Y_i and a MAC Z_i. To provide reasonable long-term security guarantees, a minimum security overhead 512 bits per log entry would be incurred in order to accommodate a 256-bit hash and a 256-bit MAC. We point out that storing two authentication tags per log entry in the Schneier-Kelsey scheme leads to storage inefficiency which makes it improper to be used on devices with commensurately meager resources such as sensors, implantable medical devices which need a light-weight secure logging system [10]. Leaving multiple authentication tags on the log file and lacking a unique authentication tag to provide integrity of the whole message body are also the causes of the Schneier-Kelsey scheme being vulnerable to truncation attacks. This motivates us to explore the recently

[4] Actually, the authors of the Schneier-Kelsey scheme do not mention any scan-verification (verification of individual MACs) in the paper. They only claim that verification of Z_f equals to verification of all the individual MACs.

proposed FssAgg authentication scheme to address both forward-secure integrity and storage efficiency.

The same set of vulnerabilities/limitations are also applicable to the Bellare-Yee private key based scheme [2,3]. The Holt scheme [9] uses a public key based approach. Therefor it avoids using a online server and also the delayed detection attack. However it still suffers from truncation attack and storage inefficiency.

3 Forward Secure Sequential Aggregate Authentication

In this section, we briefly introduce the components of a FssAgg scheme as they will be used in our secure logging system. We refer to [11, 12] for a more formal definition of a FssAgg scheme. We next show how a FssAgg scheme can provide *forward-secure stream integrity*.

An FssAgg scheme includes the following components:

[*FssAgg.Kg*] – key generation algorithm used to generate public/private key-pairs. It also takes as input T – the maximum number of time periods (key evolvements).
[*FssAgg.Asig*] – sign-and-aggregate algorithm which takes as input a private key, a message to be signed and a signature-so-far (an aggregated signature computed up to this point). It computes a new signature on the input message and combines it with the signature-so-far to produce a new aggregate signature. The final step in *FssAgg.Asig* is a key update procedure *FssAgg.Upd* which takes as input the signing key for the current period and returns the new signing key for the next period (not exceeding T). We make key update part of the sign-and-aggregate algorithm in order to obtain stronger security guarantees (see below).
[*FssAgg.Aver*] – verification algorithm, which, on input of a putative aggregate signature, a set of presumably signed distinct messages, and a public key, outputs a binary value indicating whether the signature is valid.

A secure FssAgg scheme must satisfy the following properties:

1. *Correctness:* Any aggregated signature produced with *FssAgg.Asig* must be accepted by *FssAgg.Aver*.
2. *Forward secure aggregate unforgeability:* No one, even knowing the current signing key, can make a valid FssAgg forgery.

The forward secure aggregate unforgeability implies two things. First, it is append-only - no one can change any message generated before the compromise, which further implies a FssAgg signature can provide integrity protection for the whole message body. An attacker who compromises a signer has two choices: either it includes the intact aggregate-so-far signature in future aggregated signatures, or it ignores the aggregate-so-far signature completely and start a brand new aggregated signature. What it cannot do is selectively deleting components of an already-generated aggregate signature. This append-only property resembles the property of special write-only disk used in traditional log systems. Second it is hard to remove a component signature without knowing it - so it is resistant to deletion (including truncation) attack. They are two very useful properties and we will exploit them in our applications.

We claim that FssAgg authentication implies forward-secure stream integrity, i.e.:

Forward Security: In a FssAgg scheme, secret signing key is updated through a one-way function. An attacker is thus unable to recover previous keys from the current (compromised) key and therefore unable to forge signatures from prior intervals.[5]
Stream Security. The sequential aggregation process in an FssAgg scheme preserves the order of messages so that it provides stream security; thus, re-ordering of messages is impossible.
Integrity. Any insertion of new messages as well as modification and deletion of existing messages will render the final aggregate unverifiable.

Armed with this implication, we can now construct a secure logging system from any FssAgg authentication scheme.

4 Private-Verifiable Scheme

We now describe a private-verifiable scheme that provides *forward-secure stream integrity* for audit logs. In a private-verifiable scheme, verifiers are drawn from a small "private" group. Our scheme is based on the FssAgg MAC scheme proposed in [11]. *Forward-secure stream integrity* is inherited from the FssAgg MAC scheme. To avoid an online server, two FssAgg MACs are computed over the log file with different initial signing keys. A semi-trusted verifier can only verify one of them. The other MAC is used by the trusted server to finally validate the log file. No one – including the semi-trusted verifier – can alter the contents of the log file without being detected.

Below, we present the trust model and our system assumption. Next we give details of the system operations. On top of *forward-secure stream integrity*, we add operations to start/open and close a log file such that total deletion and abnormal stop attacks can be detected. We then evaluate the proposed scheme.

4.1 Security and System Model

There are three types of players in our scheme:

1. $\mathcal{U}$ is an *untrusted* log generator. By "untrusted", we mean it is not physically secure, bug-free, or sufficiently tamper-resistant to guarantee that it can not be taken over by an attacker. $\mathcal{U}$ itself does not behave maliciously unless controlled by the attacker. It generates log entries and replies to $\mathcal{V}$'s query. It only interacts with $\mathcal{T}$ to start a log file or after a log file is closed.
2. $\mathcal{V}$ is a *semi-trusted* verifier that reads and verifies the log file on $\mathcal{U}$. Usually, audit logs can only be accessed by a small group of people, such as system administrators, security personnel and auditors. Therefore, $\mathcal{V}$ is drawn from a small group of authorized entities; it can obtain and verify a copy of the audit log from $\mathcal{U}$, when necessary. However, $\mathcal{V}$ is not trusted as far as the integrity of the log file.
3. $\mathcal{T}$ is a *trusted* machine in a secure location. It has secure storage ample enough to hold audit logs from $\mathcal{U}$. It can authorize a legitimate verifier $\mathcal{V}$ to get access to the audit log and gives $\mathcal{V}$ the verification key. It also finally validates the log file. $\mathcal{T}$ does not interfere the verification process.

[5] Assuming, of course, that the plain signature scheme – upon which the FssAgg scheme is built – is CPA-secure.

As in [5], we assume that there is no constantly available reliable high-bandwidth channel between $\mathcal{U}$ and trusted storage on $\mathcal{T}$. Consequently, $\mathcal{U}$ and $\mathcal{T}$ communicate infrequently.

The attacker's goal is to tamper with the log file by deleting, modifying, inserting or re-ordering log entries. Clearly, the attacker who compromises $\mathcal{U}$ obtains the signing key used at the time of compromise. We consider two types of attackers: outsiders and insiders. An outsider is an attacker that knows none of $\mathcal{U}$'s secrets before compromising $\mathcal{U}$. A malicious $\mathcal{V}$ is considered to be an insider attacker as it knows some of $\mathcal{U}$'s secrets. An insider is obviously more powerful as far as its ability to tamper with the integrity of the log file. Our scheme is designed to detect both insider and outsider attacks.

4.2 Description of the Scheme

We use the following notation from here onwards:

- L_i: i-th message, i.e., the i-th log entry. (We assume that log entries are time-stamped and generally have a well-defined format).
- $\mathcal{F}$: k-bit full-domain hash function with strong collision resistance $\mathcal{F} : \{0,1\}^k \rightarrow \{0,1\}^k$.
- $\mathcal{H}$: one-way hash function with strong collision resistance and arbitrarily long input: $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^k$.
- mac: secure MAC function $mac : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^t$ that, on input of a k-bit key x and an arbitrary message m, outputs a t-bit $mac_x(m)$.
- UPD: key update frequency (see below).

At any given time, an authenticated log file in this scheme consists of two parts: (1) log entries: $[L_1, \cdots, L_i]$ and (2) two authentication tags (forward-secure aggregate MACs): $\mu_{\mathcal{T},i}$ and $\mu_{\mathcal{V},i}$ that are defined below.

Log File Initialization Before the logging system starts, we require $\mathcal{T}$ to be accessible to $\mathcal{U}$ and assume that $\mathcal{U}$ is not compromised (yet). $\mathcal{U}$ generates two random symmetric keys, A_1 and B_1. Then, it commits these keys to $\mathcal{T}$ along with the other information about the specific log file and the key update interval UPD. We are not concerned with the details of the commitment process. Suffice it to say that, after the commitment process, $\mathcal{T}$ can go off-line and $\mathcal{U}$ can be deployed in an adversarial and unattended environment.[6]

Meanwhile, $\mathcal{U}$ creates the initial "dummy" log entry L_1 which commits to a fixed message (e.g., set to "START") and computes two MACs on L_1 with keys A_1 and B_1, respectively: $\mu_{T,1} = mac_{A_1}(L_1)$ and $\mu_{V,1} = mac_{B_1}(L_1)$. Next, $\mathcal{U}$ evolves its keys through a one-way function $\mathcal{F}$: $A_2 = \mathcal{F}(A_1)$, and $B_2 = \mathcal{F}(B_1)$.

Through the initial interaction, $\mathcal{T}$ knows that $\mathcal{U}$ has started a log file at time t with initial secrets A_1 and B_1. $\mathcal{T}$ stores these values in its database and thereafter knows that a valid log must exist on $\mathcal{U}$ and that that log must contain at least one log entry L_1. The purpose of this initial commitment step is to prevent a total deletion attack, i.e.,

[6] We also assume that the initial commitment as well as each subsequent log entry contains a time-stamp.

an attacker breaking into $\mathcal{U}$ at a later time cannot delete the whole log file and simply claim that no such log file has started yet.

Update Frequency We make no assumptions about key update frequency except that it must be fixed at log initialization time by $\mathcal{T}$ or $\mathcal{U}$ (or both). Moreover, it must be encoded in the first message from $\mathcal{U}$ to $\mathcal{T}$. UPD can be based on time (e.g., every hour), volume of activity (e.g., every 10 log entries) or some combination thereof. However, to simplify our discussion below, we assume that keys are updated for each log entry.

Generating Log Entries Before the i-th log entry is generated, the log file contains $L_1, \cdots, L_{i-1}$ and two FssAgg MACs $\mu_{T,i-1}$, $\mu_{V,i-1}$. Current keys of $\mathcal{U}$ are: A_i and B_i. Now, a new i-th event occurs and $\mathcal{U}$ creates a corresponding log entry L_i. $\mathcal{U}$ updates[7] authentication tags as follows:

> $\mathcal{U}$ first generates a MAC for $\mathcal{V}$ as: $mac_{A_i}(L_i)$. It then computes $\mu_{V,i}$ as: $\mu_{V,i} = \mathcal{H}(\mu_{V,i-1}||mac_{A_i}(L_i))$. Here, $\mathcal{H}$ acts as the aggregation function. Note that $\mu_{V,i}$ can be represented (un-rolled) as:
>
> $$\mu_{V,i} = \mathcal{H}(\mathcal{H}(\cdots \mathcal{H}(\mu_{V,1}||mac_{A_1}(L_1)) \cdots)||mac_{A_i}(L_i)) \quad (1)$$
>
> $\mathcal{U}$ updates the second FssAgg MAC (for $\mathcal{T}$) in the same manner: $\mu_{T,i} = \mathcal{H}(\mu_{T,i-1}||mac_{B_i}(L_i))$ Finally, $\mathcal{U}$ evolves both keys: $A_{i+1} = \mathcal{F}(A_i)$, and $B_{i+1} = \mathcal{F}(B_i)$. Prior keys A_i and B_i and MACs $mac_{A_i}(L_i)$ and $mac_{B_i}(L_i)$ are immediately and securely erased (e.g., from disk and RAM).

Log File Closure $\mathcal{U}$ officially closes the log file by creating a special closing message as the final log entry (L_f), updating the two authentication tags ($\mu_{V,f}$ and $\mu_{T,f}$) and securely erasing the remaining keys (A_f and B_f).

The closing step is necessary in order to inform users that the log file is closed properly and no longer accepts any new data. Consider that an attacker might prevent the logging system from functioning after gaining control of the logging machine. Without the explicit closing step, we can not determine whether the log file has been closed normally or the logging process has been impeded by an attacker. Once the log file has been properly closed, an attacker who breaks into $\mathcal{U}$ cannot modify the log file since no keys are available.

Log File Validation An authorized verifier $\mathcal{V}$ starts the log file validation process by obtaining A_1 – one of the two initial signing keys – from $\mathcal{T}$. Next, $\mathcal{V}$ queries $\mathcal{U}$ and obtains a copy of log entries $L_1, \cdots, L_f$ as well as $\mu_{V,f}$. $\mathcal{V}$ computes $A_2, \cdots, A_f$ through the key update function, computes $\mu'_{V,f}$ and checks that it matches $\mu_{V,f}$. Verifier's computation costs amount to f invocations of $\mathcal{F}$, $\mathcal{H}$ and mac.

When $\mathcal{T}$ receives the complete and closed log file, it can independently validate it using B_1 and $\mu_{T,f}$. The validation mimics that performed by $\mathcal{V}$. Note that, a malicious verifier $\mathcal{V}$, knowing A_1, has full control and can modify any log entries by generating its own version of $\mu_{V,f}$. However, it can not forge $\mu_{T,f}$.

[7] We use the term "updates", since, at all times, there are only two authentication tags in the secure log.

4.3 Discussion

The private-verifiable scheme described above is simple and very computation-efficient, since it involves fast hashing and symmetric key operations. $\mathcal{V}$ can verify a log file without consulting $\mathcal{T}$; thus, no on-line trusted party is needed. Furthermore, it is very storage-efficient: compared with previous schemes – which require either f or $2 * f$ storage units to store authentication incurred values – our scheme only needs two storage units for two FssAgg MACs. Considering that log files tend to be very large and often contain many thousands of log entries, the benefit of a storage-efficient scheme is quite apparent.

Our scheme provides *forward-secure stream integrity* through the use of a single FssAgg MAC that covers all log entries. An attacker can not forge such a MAC without knowing any pre-compromise MAC keys. Deletion and truncation attacks can be detected by any verifier. Furthermore, our scheme can detect a total deletion attack since we use an explicit commitment process when starting a log file. By explicitly closing the log file, our scheme can detect certain DoS attacks that aim to incapacitate the logging system.

However, a malicious verifier $\mathcal{V}$ can tamper with the log without being detected by other verifiers. This tampering can only be detected with the help of $\mathcal{T}$. It is thus possible for a malicious insider to mount a delayed detection attack. This is a definite drawback which leads us to construct an alternative scheme based on public key techniques.

5 A Public-Verifiable Scheme

We now describe a public-verifiable scheme. It can be based on either the BLS-$FssAgg$ scheme proposed in [11] or the BM-$FssAgg$ scheme we proposed in Section 3. A public-verifiable scheme allows auditors outside the system to make sure no tampering takes place within the system. Therefore, it can be used for systems which need to be publicly audited, such as financial records for public companies and voting systems in democratic countries. A public-verifiable scheme also avoids the shortcoming of a private-verifiable scheme which still suffers from delayed detection attacks whenever a private verifier $\mathcal{V}$ behaves maliciously.

As in the last section, we begin with the trust model and system assumptions. Next, we describe the scheme evaluating it. For the sake of brevity, we focus on the difference between private- and public-verifiable schemes.

5.1 Trust Model

In this scheme we no longer require a trusted server $\mathcal{T}$. Instead, we need a Certification Authority (CA) that can certify/register $\mathcal{U}$'s public key. The scope of $\mathcal{V}$ moves from a small private group of semi-trusted entities to the public domain, i.e., anyone who has a copy of the log file can verify it. We no longer need to differentiate an inside attacker and outside attackers. An attacker is thus anyone who behaves maliciously and does not know the system's initial secrets.

5.2 Scheme Description

An authenticated log file in the present scheme consists of two parts: log entries $[L_1, \cdots, L_f]$ and a single FssAgg signature, $\sigma_{1,f}$.

Log File Initialization To initiate a log file, $\mathcal{U}$ uses $FssAgg.Kg$ to generate the initial secret key sk_1 and the public key pk. Then it registers pk with a public CA. $\mathcal{U}$'s certificate for log file should contains at least these essential information such as the log creator, the log ID, starting time and the public key. For example, CA's signature in $\mathcal{U}$'s certificate for log file ID_{log} may look like:

$$CERT(ID_{log}) = SIGN_{CA}(\mathcal{U}, ID_{log}, t, T, pk, \text{timestamp}, \cdots)$$

$\mathcal{U}$ keeps sk_1. Next, it creates the initial log entry L_1 which it sets to the certificate $CERT(ID_{log})$. Then, $\mathcal{U}$ generates a signature $\sigma_{1,1}$ on L_1 with $FssAgg.Asig$ using the initial private key sk_1. Finally, $\mathcal{U}$ updates its key from sk_1 to sk_2 and securely erases any and all copies of sk_1.

Generating Log Entries Before the i-th entry occurs, the log file contains $[L_1, \cdots, L_{i-1}]$ and a FssAgg signature $\sigma_{1,i-1}$. $\mathcal{U}$'s current secret key is sk_i. Now, a new event occurs and triggers $\mathcal{U}$ to creates a new log entry L_i. $\mathcal{U}$ updates the FssAgg signature by inputting function $FssAgg.Asig$ with parameters: L_i, $\sigma_{1,i-1}$ and the current key sk_i. Finally, $\mathcal{U}$ evolves its private key through the update function $FssAgg.Upd$ and securely erases sk_i. (In our context, the key update is invoked immediately after the aggregate signature is generated.)

As the log file grows – since the maximum number of key update periods T is fixed *a priori* – the number of updates might need to exceed T. To address this issue we can dynamically extend the scheme to support additional key update periods without sacrificing security. One straightforward way is to generate a public key for the next T number of time periods and to use the last (initially certified) secret key sk_T to, in turn, certify a new set of public keys to be used subsequently. In fact, the certification of the next batch of public keys should be treated as a special log entry L_T.

Log File Closure As in the private-verifiable scheme, $\mathcal{U}$ closes the log file by creating a special closing message as the final log entry L_f, updating the FssAgg signature accordingly, and securely erasing its secret key.

Validating Log File After receiving a copy of the log file, $\mathcal{V}$ extracts public keys from $CERT(ID_{log})$ contained in the initial log entry L_1 and $\mathcal{V}$ verifies CA's signature on $CERT(ID_{log})$. Then, $\mathcal{V}$ validates the actual log file using the aggregate function $FssAgg.Aver$.

5.3 Discussion

Compared with the private-verifiable scheme, the present scheme provides better security because of its resistance to delayed detection attacks. It allows anyone – not just a group of semi-trusted verifiers – to validate a log file. It is thus suitable for applications where scalability is important and, more generally, where public verification is required. Except for the log initialization time, no trusted entity is needed for any system operations.

6 Immutable Forward-Secure Aggregate Authentication

In Section 4 and 5, we proposed secure logging systems that are very efficient in term of storage and communication - only ONE aggregate tag is kept in the audit log and individual tags are erased once they are folded into the aggregate. So they are very suitable to be used on devices with meager resources, especially storage, as such devices require a light-weight audit log system. Verification of individual log entry is implied by the verification of the entire log file.

However, this indirect verification is very costly if only one particular log entry is interested. For example, users of a versioning file system might be only interested in one particular version [13]. Verifying all versions is computationally expensive (especially, in a public-verifiable scheme) and involves transferring unwanted data (i.e., all other versions) to the user. Furthermore aggregate verification failure does not tell the user anything about the authenticity of this particular version - there might be something wrong with other versions. It is also desirable to keep individual authentication tags in some applications. For example, in outsourced audit log applications [14, 15] where audit logs from different log servers are submitted to a third party repository, an auditor may want to search log entries satisfying certain properties, such as destination address or port number. Log entries in the query result set may come from different log servers. In this scenario, we need individual signatures so that the untrusted repository can use the techniques proposed in [16] to answer queries.

This motivates us to keep individual signatures in the log file - in applications where storage is not a problem - to provide finer grained verification. The aggregate signature is used to protect the integrity of the whole log file while individual signatures are used for individual log entry verification. However, the aggregation functions in all aforementioned FssAgg schemes are public. Thus, revealing individual signatures or MACs allows anyone to create new authentic aggregate signatures or MACs.

For example, suppose an attacker obtains a copy of log entries $[L_1, \cdots, L_f]$, their corresponding individual authentication tags as well as the aggregate authentication tag. In our private-verifiable scheme, the attacker can truncate the log file from $(i+1)$-st log entry and then compute a new valid aggregate tags $\mu_{T,i}$ and $\mu_{V,i}$ using Eq. 1 for log entries $[L_1, \cdots, L_i]$ $(i < f)$. In our public-verifiable scheme, the attacker can similarly generate a new aggregate signature by removing $[\sigma_{i+1}, \cdots, \sigma_f]$ from $\sigma_{1,f}$ by computing: $\sigma_{1,i} = \sigma_{1,f} / \prod_{j=i+1}^{f} \sigma_j$.

To prevent truncation attacks when individual component signatures are revealed, we need immutable FssAgg authentication schemes. We note that mutability is not a flaw of the underlying FssAgg signature schemes but rather an issue with some specific audit log applications. The immutability of an immutable FssAgg authentication scheme in our specific log scenario means the difficulty of computing new valid FssAgg signatures from a set of other aggregate signatures (an individual signature σ_i can be regarded as an aggregate signature $\sigma_{i,i}$). We note the difference between a FssAgg signature and a normal aggregate signature: a FssAgg signature for a log file is bound to a unique initial public key and covers a set of continuous log entries starting from L_1, while an aggregate signature may cover isolated log entries. For example, $\sigma_1 \cdot \sigma_2 \cdot \sigma_3$ (where each σ_i is generated with a different secret key) is an FssAgg signature for a log file containing three log entries, while $\sigma_2 \cdot \sigma_3$ is just an aggregate signature of the last two log entries.

Immutable aggregate signatures have been previously considered in [17] to prevent computation of a new aggregate signature $\sigma_{j,k}$ when one "sees" both $\sigma_{i,j}$ and $\sigma_{i,k}$ where $i < j < k$. However this is not a problem in our envisaged log applications since such a signature, although it might be a valid aggregate signature for log entries from L_j to L_k, is no longer a valid FssAgg signature for a log file bound to a fixed initial verification key.

6.1 Immutable FssAgg MAC Scheme

The immutability extension for the private-verifiable FssAgg MAC scheme is very simple: $\mathcal{U}$ generates a "phantom" MAC and places it as the first component into the aggregate when the system starts; this "phantom" MAC is then erased right after aggregation. We modify the log file initialization (in Section 4) as follows:

1. $\mathcal{U}$ computes $\mu_1 = mac_{A_1}(L_1)$ over a fixed (well-known) initialization message L_1;
2. $\mathcal{U}$ evolves A_1 into A_2;
3. When the first real log entry L_2 occurs, $\mathcal{U}$ generates $\mu_2 = mac_{A_2}(L_2)$ and aggregates: $\mu_{V,2} = \mathcal{H}(\mu_1 || \mu_2)$;
4. $\mathcal{U}$ stores both μ_2 and $\mu_{V,2}$ in the log file, evolves A_2 into A_3, securely erases μ_1 and A_2, and officially moves to next time period.

Note that we do not change anything involving the use of B_1 and subsequent keys B_i involved in the generation of MACs for T. In other words, only the computation of $\mathcal{V}$-verifiable values is altered.

The resulting format of the log file supporting both immutability and (efficient) individual log entry verification is:

$$\{[L_0, (L_1, \mu_1), (L_2, \mu_2), \cdots, (L_f, \mu_f)],\ \mu_{V,f}, \mu_{T,f}\}$$

If $\mathcal{V}$ is only interested in verifying a particular log entry L_i, it uses μ_i directly without involving the aggregate $\mu_{V,f}$. However, note that, to verify L_i, $\mathcal{V}$ still needs to recompute A_i which requires i hash operations.

An attacker who obtains all log entries with their *exposed* MACs: $[\mu_1, \cdots, \mu_f]$ and the FssAgg MAC $\mu_{V,f}$ is unable to compute a new valid FssAgg MAC without the knowledge of μ_0. The security of this extension relies on the pre-image resistance of the underlying hash function.

6.2 Immutable FssAgg Signature Scheme

The iBGLS scheme in [17] achieves immutability by combining a signature of the database server with the aggregate-so-far BGLS signature of the data owner. Different query result sets and different queriers lead to distinct server signatures. The server signature is computed over all the messages in the query result set and thus acts as an "umbrella" signature. This "umbrella" signature is never revealed to public. It is impossible to remove such a signature since it is hard to remove a component signature without knowing it - a property of aggregate signature schemes.

In our context, we still exploit this property and use the same idea by folding into the aggregate an "umbrella" signature generated over all current log entries. However, since our security model assumes eventual compromise of $\mathcal{U}$ we must fold umbrella signatures into the aggregate before an attacker breaks in. We define the time period at which an umbrella signature is generated and aggregated as an *anchor point* and the corresponding log entry generated in this time period is called as an *anchor log entry*.

For each anchor log entry $L_j, \mathcal{U}$ computes two signatures: a normal individual signature σ_j over L_j, and an umbrella signature σ_j^* over $k+1$ log entries of $L_{j-k}, \cdots, L_j$. The umbrella signature σ_j^* is aggregated with the aggregate-so-far signature. After the aggregation, σ_j^* is securely erased. σ_j is kept in the log file for individual log entry verification. For non-anchor-log entries, $\mathcal{U}$ proceeds as usual except that now individual signatures are stored in the log file (instead of being deleted after aggregation). Let $\mathcal{L}$ denote the set of normal log entries and $\mathcal{L}^*$ the set of anchor log entries. An immutable FssAgg signature can be represented as:

$$\sigma_{1,f} = \prod_{L_i \in \mathcal{L}} \sigma_i \prod_{L_j \in \mathcal{L}^*} \sigma_j^* \qquad (2)$$

So far, three FssAgg signature schemes, the BLS-$FssAgg$ scheme [11], the BM-$FssAgg$ scheme and the AR-$FssAgg$ scheme [12], have been proposed. Now we show how to extend them to provide immutability.

It is easy to modify the BLS-$FssAgg$ scheme to support immutability. σ_j is now computed as:

$$\sigma_j = \mathcal{H}_1(index||L_j)^{x_j}$$

and σ_j^* is computed as:

$$\sigma_j^* = \mathcal{H}_1(index||L_{j-k}||L_{j-k+1}||\cdots||L_j)^{x_j}$$

Constructing an immutable version of BM-$FssAgg$ is a little more involved. First, during the key generation stage, we select two random values r_0, r_0^* and generate two common commitments: $y = (r_0)^{2^{T+1}}$ and $y^* = (r_0^*)^{2^{T+1}}$. We use r_j to generate a normal signature (σ_j) and r_j^* to generate an umbrella signature (σ_j^*) . σ_j is computed as:

$$\sigma_j = r_j \prod_{i=1}^{l} s_{i,j}^{c_i} \quad (c_1 \cdots c_l \leftarrow H(t, y, L_j))$$

and σ_j^* is computed as:

$$\sigma_j^* = r_j^* \prod_{i=1}^{l} s_{i,j}^{c_i} \quad (c_1 \cdots c_l \leftarrow H(t, y, L_{j-k}, \cdots, L_j))$$

An authenticated log file supporting immutability and individual log entry verification has the following format:

$$\{[(L_1, \sigma_1), (L_2, \sigma_2), (L_3, \sigma_3), \cdots, (L_f, \sigma_f)], \sigma_{1,f}\}$$

where $\sigma_{1,f}$ is computed as in Eq. 2. The security of the modified schemes is implied by the property of aggregate signature scheme: it is computationally hard to remove a component signature without knowing it.

We can use the same idea to construct an immutable version of AR-$FssAgg$ signature scheme. We omit the details because of page limitation.

7 Evaluation

We evaluate our secure logging scheme by comparing it with the existing schemes. We compare our private verifiable scheme with two existing private-key-based schemes: Schneier-Kelsey [5] and Bellare-Yee [3]. We also compare our public-verifiable scheme with Holt's scheme [9]. Our comparison is based on four factors: 1) resilience to truncation attacks; 2) resilience to delayed detection attacks; 3) on-line server requirements; 4) storage efficiency. The comparison results are summarized in Table 7.

Table 1. Comparisons of Various Schemes.

	Private Key Based Schemes			**Public Key Based Schemes**	
	SK [5]	BY [3]	Ours	Holt [9]	Ours
Resilience to truncation attack?	No	No	Yes	No	Yes
Resilience to delayed detection attack?	No	No	No	Yes	Yes
No on-line server?	No	No	Yes	Yes	Yes
Storage efficient?	No	No	Yes	No	Yes

Compared with Schneier-Kelsey and Bellare-Yee, our private scheme is resilient to truncation attacks, more storage-efficient and requires no on-line server. However, it is still vulnerable to delayed detection attacks. Compared with Holt's scheme, our public scheme is resilient to truncation attacks and more storage-efficient.

8 Implementation

We investigated the viability of our proposed schemes on a Intel dual-core 1.73GHz Laptop with 1GB RAM running Linux. We used the NTL library [18] and the PBC library [19] as the underlying cryptographic libraries.

We built our prototype based on the code from the OpenCM project [20], a free source version control software. OpenCM is a client/server application whereby individual developers typically work on their own workstations with the repository hosted on a server. As with other versioning systems, such as CVS [21], our system (see Figure 2) allows authorized developers (through access control) to check out a baseline version of the software, make modifications, and commit the result as the new state of the system. When a developer's work is to be deposited into the repository, the server checks

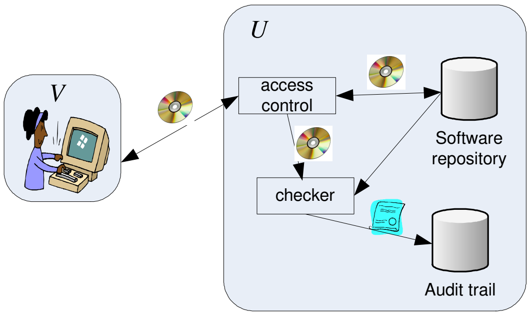

Fig. 2. Secure Version Control System.

any changes between the new and old versions and then creates a log entry recording the modification into a log file. Our schemes are incorporated into the server-side component in order to provide accountable verification of the audit trail.

In the implementation of our private-verifiable scheme, there is no separate physical $\mathcal{T}$ component to which $\mathcal{U}$ transmits its closed log file. Instead, a logical $\mathcal{T}$, periodically validates the audit trail directly on $\mathcal{U}$. $\mathcal{T}$ does not hold a copy of the audit trail.

We prototyped all the three FssAgg signature schemes to provide heuristics for choosing among them in practice. For the BM-$FssAgg$ and AR-$FssAgg$ schemes, we selected security parameters $k = 1024$ and $l = 160$. For the BLS-$FssAgg$ scheme, we used a singular curve $Y^2 = X^3 + X$ defined on a field F_q for $|q| = 512$ and the group order $|p| = 160$, where p is a Solinas prime. Such a group has the fastest pairing operations [19]. We measured signer's computation cost by signature generation and key update on a per-log-entry basis. We measured verifier's computation cost over an aggregate signature $\sigma_{1,t}$ when $t = 100, 1,000$ and $10,000$ which corresponds to a small, medium, and large log file, respectively. Experimental results shown in Table 8 show that the BM-$FssAgg$ scheme is the most efficient one in terms of computation (for both signer and verifier). Its signature generation is approximately 2 times faster than that of the AR-$FssAgg$ and 5.5 times faster than that of the BLS-$FssAgg$. Its signature verification is 4 times faster than that of the AR-$FssAgg$ and 16 times faster than that of the BLS-$FssAgg$. However, it incurs the most storage overhead.

We also investigated storage overhead incurred by each scheme. Let I_a denote the amount of storage needed to to store the secret key and the aggregate signature - the overhead incurred by authentication. Let $|S|$ denote the size of a signature or a key. Let I denote the number of log entries and $|L|$ denote the average size of a log entry. We measure storage efficiency by $\frac{I_a * |S|}{I * |L|}$. BLS-$FssAgg$ needs 1 unit of space each for both secret key and signature. BM-$FssAgg$ needs 162 units of space for secret key and 1 unit of space for the aggregate signature. BM-$FssAgg$ needs 2 units of space for secret key and 1 unit of space for the aggregate signature. To simply our measurement, we assume a log entry size is comparable to the size of a signature or a secret key, e.g. $|S| \approx |L|$. The comparison result is also shown in Table 8. BLS-$FssAgg$ performs best in term of storage efficiency. Especially when there is a large number of log entries and each entry is large, storage overhead of BLS-$FssAgg$ is negligible.

Table 2. Comparisons of FssAgg Signature Schemes. (Operation Timing in *msecs*.)

		BLS-$FssAgg$	**BM-**$FssAgg$	**AR-**$FssAgg$
Signer Computation Cost (per log entry)	$Asig$	30	2.09	4.39
	Upd	0.002	3.46	7.27
	total	30.00	5.55	11.66
Signer Storage Cost	$t = 100$	2%	162%	3%
	$t = 1000$	0.2%	16.2%	0.3%
	$t = 10000$	0.02%	1.62%	0.03%
Verifier Cost	$t = 100$	3.30×10^3	211.97	810.88
	$t = 1000$	29.3×10^3	2.13×10^3	8.16×10^3
	$t = 10000$	330.72×10^3	21.35×10^3	80.84×10^3

9 Related Work

A number of cryptographic approaches to address secure logging have been proposed to-date. Most prior work focused on three areas: (1) data integrity/authentication, (2) data confidentiality and access control, and (3) searchable encryption. Since we are primarily interested in integrity, only the first area directly relates to this paper. Bellare and Yee were the first to define the *forward-secure stream integrity* property required in an audit log system and proposed to use forward-secure MACs [2, 3]. They focused on formal definition and construction of forward-secure MAC schemes and applied them to secure audit log applications. In their secure log scheme, multiple log entries are indexed and tagged independently within one time period. At the end of each time period, a special log entry containing the number of log entries in the current time period is created to indicate the end of the current time period. This scheme has the same security as well as the architectural limits as the SK scheme. Schneier and Kelsey proposed a similar system (the SK scheme we analyzed in Section 2) based on forward-secure MAC and one-way hash chains [5, 22, 23]. Unlike Bellare and Yee's scheme, in the SK scheme, rekeying is performed after each log entry is made. Therefore they no longer use per-stage sequence numbers in tagging logs. Instead, each log entry now contains a link in a hash chain and a forward-secure MAC computed over this link to authenticate the values of all pervious entries. Moreover, they presented a precise protocol design for its implementation in a distributed system, describing how messages are sent to external trusted machines upon log creation and closing. Chong, et. al. discussed the feasibility of using of tamper-resistent hardware in conjunction with a system like Schneier and Yee's in [24]. Holt extended Schneier and Yee's system to the public key setting [9]. Waters, et. al. designed encrypted and searchable audit log [25]. This showed how identity-based encryption (IBE) can be used to make audit logs efficiently searchable. Keywords which relate to each log entry are used to form public keys in an IBE system. Administrators allow searching and retrieval of entries matching a given set of keywords by issuing clients the corresponding IBE private keys. They recommended the use of the Schneier and Yee's technique as their authentication scheme. The two security attacks, truncation attack and delayed detection attack, which we outlined in Section 2, seem to be very fundamental to all the secure audit log schemes as far as we know. It is surprising that they have not been addressed by any later work so far.

10 Conclusion

In this paper, we identified some issues in current secure logging techniques. We then proposed two concrete schemes to provide *forward-secure stream integrity* for logs generated on untrusted machines. Our approach supports forward security and compact aggregation of authentication tags (MACs or signatures). Both of our proposed schemes offer practical secure logging without reliance on trusted third parties or secure hardware. Our schemes are based on the recent proposed FssAgg authentication schemes where a unique authentication tag is used to protect the integrity of underlying message body. We then considered the notion of immutability that is needed to facilitate faster verification of individual log entries. We evaluated the performance of our schemes and report on experience with a prototype implementation within a public domain versioning control system.

Although the security of proposed schemes rests entirely on previously proposed techniques (i.e., [11,12] and [17]), we need to construct separate security proofs for each scheme. Furthermore, we have to conduct extensive experiments, and perhaps trace-driven simulations, to better understand the performance of our schemes. Finally, we intend to investigate alternative signature schemes that might be used for constructing more efficient public-verifiable techniques.

References

1. Swanson, M., Guttman, B.: Generally accepted principles and practices for securing information technology systems. In: NIST 800-14. (1996)
2. Bellare, M., Yee, B.: Forward integrity for secure audit logs. In: Technical Report, Computer Science and Engineering Department, University of San Diego. (November 1997)
3. Bellare, M., Yee, B.: Forward-security in private-key cryptography. In: Proc. of CT-RSA'03. (2003)
4. U.S. Department of Defense, C.S.C.: Trusted computer system evaluation criteria (December 1985)
5. Schneier, B., Kelsey, J.: Cryptographic support for secure logs on untrusted machines. Proceedings of the 7th USENIX Security Symposium (Jan. 1998)
6. Bellare, M., Palacio, A.: Protecting against key exposure: strongly key-insulated encryption with optimal threshold. In: Cryptology ePrint Archive, Report 2002/64. (2002)
7. Dodis, Y., Katz, J., Xu, S., Yung, M.: Key-insulated public key cryptosystems. In: Eurocrypt'02. (May 2002) 65–82
8. Dodis, Y., Katz, J., Xu, S., Yung, M.: Strong key-insulated public key cryptosystems. In: Public Key Cryptography (PKC 2003). (April 2003) 130–144
9. Holt, J.E.: 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, Darlinghurst, Australia, Australian Computer Society, Inc. (2006) 203–211
10. Halperin, D., Kohno, T., Heydt-Benjamin, T., Fu, K., Maisel, W.: Security and privacy for implantable medical devices. **7**(1) (January 2008)
11. Ma, D., Tsudik, G.: Forward-secure sequentical aggregate authentication. In: Proceedings of IEEE Symposium on Security and Privacy 2007. (May 2007)
12. Ma, D.: Practical forward secure sequential aggregate signatures. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS'08). (March 2008)

13. Burns, R., Gentry, C., Lynn, B., Shacham, H.: Verifiable audit trails for a versioning file system. In: StorageSS'05. (Nov 2005) 416–432
14. Symantec: Symantec deepsight threat management system technology brief https://tms.symantec.com/.
15. : The dshield project. http://www.dshield.org
16. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. In: NDSS 2004. (February 2004)
17. Mykletun, E., Narasimha, M., Tsudik, G.: Signature bouquets: immutability for aggregated/codensed signatures. In: ESORICS'04. (2004) 160–176
18. Shoup, V.: Ntl: a library for doing number theory. http://www.shoup.net/ntl/
19. : Pbc library benchmarks. http://crypto.stanford.edu/pbc/times.html
20. : Opencm project. http://www.opencm.org
21. Berliner, B.: Cvs ii: parallelizing software development. In: Proc. of the USENIX Winter 1990 Technical Conference. (1990) 341–351
22. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Transactions on Information and System Security (1999) 159–176
23. Kelsey, J., Schneier, B.: Minimizing bandwidth for remote access to cryptographically protected audit logs. In: Recent Advances in Intrusion Detection (RAID99). (Sep. 1999)
24. Chong, C., Peng, Z., Hartel, P.: Secure audit logging with tamper resistant hardware. In: Technical report TR-CTIT-02-29, Centre for Telematics and Information Technology, Univ. Twente, The Netherlands. (August 2002)
25. Waters, B., Balfanz, D., Durfee, G., Smeters, D.K.: Building an encrypted and searchable audit log. In: ACM Annual Symposium on Network and Distributed System Security (NDSS'04). (2004)