# Practical Forward Secure Sequential Aggregate Signatures

Di Ma
Computer Science Department
University of California, Irvine
dma1@ics.uci.edu

## ABSTRACT

A forward secure sequential aggregate ($FssAgg$) signature scheme allows a signer to iteratively combine signatures generated in different time intervals – and with different keys – into a single constant-size signature. Such a signature offers forward security, storage/communication efficiency, as well as overall integrity of the signed messages. $FssAgg$ schemes are therefore suitable for data-intensive applications on untrusted and/or unattended devices, e.g., logging systems. The first $FssAgg$ signature scheme [21] is based on bilinear maps and is thus rather costly. In this paper, we propose two more practical $FssAgg^1$ signature schemes. A $FssAgg^1$ signature scheme is a special $FssAgg$ signature scheme where exactly ONE message can be signed at each time interval and key update is invoked immediately after each signature generation. Both new schemes are derived from existing forward secure signature schemes. Unlike the scheme in [21], each new scheme has constant-size public and private keys, constant-size signatures as well as constant-time key update and signature generation complexity. We show how to apply proposed schemes in secure logging applications.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

## General Terms

Security

## Keywords

Forward secure sequential aggregate authentication, MAC, signature, quality of forward security

## 1. INTRODUCTION

Forward secure sequential aggregate ($FssAgg$) signatures are proposed to reconcile minimal storage/communication overhead with mitigating potential key exposure. It allows the signer to combine signatures generated in different intervals under different secret keys incrementally and sequentially in a layered "onion-like" fashion with the first signature innermost in the aggregate. In a $FssAgg$ scheme, the verifier uses a single public key to verify the entire aggregate. In contrast with general (not forward-secure) aggregate signature schemes [8, 19, 20] which aggregate signatures from *multiple* signers, a $FssAgg$ scheme aggregates signatures of a *single* signer.

$FssAgg$ authentication was first introduced in [21] and the original motivation stemming from non-networked unattended sensor scenario where a collector visits periodically and gathers accumulated data from individual sensors. In that setting, forward security mitigates effects of potential sensor compromise, while aggregation reduces storage and communication overhead. $FssAgg$ authentication is also relevant to logging systems where forward-secure authentication of multiple log entries is needed and compromise is possible. A $FssAgg$ scheme provides forward security, storage/communication efficiency as well as stream integrity for the entire log, i.e., any modification, insertion, deletion or reordering of pre-compromise log entries renders the aggregate signature unverifiable. Overall, a $FssAgg$ scheme is a perfect match for applications where *forward secure stream integrity* is needed [6]. In a secure logging system, the signer (log server) has no storage or bandwidth limitations. However, resistance to so-called *truncation attacks* is important. Informally, a truncation attack occurs when the adversary succeeds in deleting a contiguous subset of tail-end messages. Unfortunately secure logging schemes such as [7] and [24] are vulnerable to truncation attacks because of the lack of a single tag protecting integrity of the entire log. A $FssAgg$ authentication scheme is well-suited for secure logging applications – it resists truncation attacks due to its all-or-nothing (aggregate and forward-secure) signature verification.

Two $FssAgg$ schemes, one MAC-based and one signature-based, are proposed in [21]. The MAC-based scheme is near-optimal in terms of efficiency. However it (naturally) does not offer non-repudiation and public (transferrable) verification. The $FssAgg$ signature scheme (hereafter we refer it as BLS-$FssAgg$) was derived from the BLS/BGLS signature schemes [8, 9]. It is signer-efficient, but not verifier-friendly as the latter needs $O(T)$ ($T$ is the maximum number of allowed intervals) space to store the public key. Also, aggregate verification is expensive because of costly pairing operations.

The BLS-$FssAgg$ scheme was designed for sensor applications where efficient signer computation and storage are pre-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ASIACCS* '08, March 18-20, Tokyo, Japan
Copyright 2008 ACM 978-1-59593-979-1/08/0003 ...$5.00.

ferred. A sensor's limited onboard storage prevents it from accumulating many data entries between successive visits of the sink. Therefore, although verification is expensive, it is justifiable given that: (1) the verifier (who is the data collector) is a powerful machine and (2) sensors only generate data at a very low frequency and only limited number of data entries are accumulated during each collection interval. However apparently it is not suitable to be used in data-intensive applications like databases. Hence, although the proposed signature scheme can be useful, it is not practical for many other applications in its current form.

This motivates us to construct more practical schemes – with either (or both) compact public keys or lower verification complexity. Intuitively, there are two ways to construct a $FssAgg$ signature scheme: either by extending an aggregate signature scheme to be forward-secure, or by extending a forward-secure signature scheme to be aggregatable. The work in [21] takes the former approach and this paper explores the latter. We propose two practical $FssAgg$ signature schemes with a special feature: exactly one can be signed and aggregated per each key evolution interval (whereas, the BLS-$FssAgg$ scheme allows signatures and aggregations within an interval). To this end, we use the notation $FssAgg^1$ and $FssAgg^m$ to distinguish between the sign-once and sign-many schemes. The notation $FssAgg$ is used when this distinction is unnecessary. We also argue that, most relevant applications require only $FssAgg^1$ schemes.

The two new $FssAgg^1$ schemes (BM-$FssAgg^1$ and AR-$FssAgg^1$, respectively) are superior to the BLS-$FssAgg$ scheme in almost all parameters, e.g., constant public key size and efficient aggregate verification. Table 1 summarizes the asymptotic performance of our schemes in comparison with the BLS-$FssAgg$ signature scheme. (Detailed evaluation is in Section 8.) Our experiments show that aggregate verification in BM-$FssAgg^1$ and AR-$FssAgg^1$ is (respectively) 16 and 4 times faster than that in BLS-$FssAgg$.

*Contributions:* We identify a new feature termed *quality of forward security* for $FssAgg$ schemes and point out that, in practice, any application scenario using a $FssAgg^m$ scheme can – with no loss of security or functionality – use a $FssAgg^1$ scheme. We construct two practical and provably secure $FssAgg^1$ schemes that perform better than prior art. We show how new $FssAgg^1$ schemes can be used in secure logging applications to provide *forward secure stream integrity*. Finally we evaluate the performance of proposed schemes.

*Organization:* The rest of this paper is organized as follows. In Section 2 we summarize related work. Section 3 defines $FssAgg$ schemes. Section 4 discusses the *quality of forward security* feature. Next, we present the BM-$FssAgg^1$ scheme in Section 5 and the AR-$FssAgg^1$ scheme in Section 6. We then describe secure logging applications in Section 7 and evaluate the performance of both schemes in Section 8. Related issues are discussed in Section 9.

## 2. RELATED WORK

The notion of forward security was introduced in the context of key-exchange protocols [12] and later adapted to signature schemes to address the *key exposure problem* of ordinary signature schemes. Forward-secure signatures were first proposed by Anderson [4] and subsequently formalized by Bellare and Miner in [5]. Bellare and Yee examined forward security in the context of conventional cryptography in [7]. In a forward-secure signatures scheme, the forward-security property is attained by dividing time into $T$ discrete *intervals*, and using a different secret key within each interval. Each subsequent key is computed from the current key via a special *key update* process. It must be computationally hard for an adversary to compute a prior interval's key from the current key; therefore, compromise of the current key does not invalidate (or allow forgery of) signatures generated in before compormise. The main challenge in designing forward-secure signature schemes is efficiency: an ideal scheme must have constant (public and secret) key sizes, constant signature size as well as constant signing, verification, and (public and secret) key update operations.

Current forward-secure signature schemes can be divided into two categories. The first is comprised of generic constructions that can use any arbitrary base signature scheme [4,5,17,22]. The main advantage of such schemes is that they have provable security. Schemes in this category are further divided into tree [5, 22] and non-tree [4, 5, 17] constructions. Non-tree constructions are simple but have one or more parameters with size/complexity linear in $T$ [4, 5]. The first tree-based construction [5] has constant-size public key and signatures, while the secret key size, signing and verifying time are $O(log\ T)$. Later Merkle trees are applied so that signing and verifying require $O(log(T))$ hashes (instead of signing or verifying operations) [22].

The other category is comprised of schemes built upon standard signature schemes [3, 5, 15, 16]. The main advantage of these schemes is that they achieve better dependence on $T$. In particular, they typically have constant size parameters. The first such scheme is based on the Fiat-Shamir signature scheme [5]. Abdalla and Reyzin scheme [3] shortens secret and public keys of [5] at the expense of signing and verifying time. Itkis and Reyzin scheme [15] has optimal signing and verifying time derived from the underlying Guillou-Quisquater signature scheme. However, it has expensive key update. Finally, Kozlov and Reyzin construct a scheme with fast key update [16]. Recently, Boyen, et al. proposed a forward-secure signature scheme with untrusted update [10]. In it, the secret key is encrypted with a "second factor", e.g., a user's password. The secret key can be updated in its encrypted form. The second factor and the encrypted secret must be present in order to sign a message.

The present paper is also relevant to aggregate signature schemes. An aggregate signature scheme combines $k$ signatures generated by $n$ signers ($k \geq n$) into a single and compact aggregate that, if verified, simultaneously verifies every component signature. Several aggregate signature schemes have been proposed in the literature, starting with the initial seminal result by Boneh, et al. [8] based, in turn, on the BLS scheme [9] operating in groups with efficient bilinear maps. Aggregate verification of this scheme is very expensive requiring $n + 1$ pairing operations. Subsequently, Lysyanskaya, et al. proposed a sequential RSA-based aggregate scheme [20]. In such a scheme, the signature is constructed sequentially in a layered fashion. Next, Lu, et al. [19] proposed another sequential aggregate signature scheme with more efficient verification.

We note that, unlike general aggregate signature scheme, a $FssAgg$ scheme aggregates signatures from the *same* signer. However, these signatures are computed within different intervals and thus with different keys. It implies incremental

**Table 1: Performance of different schemes. $T$ denotes the maximum of intervals over which the public key is valid.**

| Parameters | BM-$FssAgg^1$ | AR-$FssAgg^1$ | BLS-$FssAgg$ |
|---|---|---|---|
| Signature Size | $O(1)$ | $O(1)$ | $O(1)$ |
| Private Key Size | $O(1)$ | $O(1)$ | $O(1)$ |
| Key Update Time | $O(1)$ | $O(1)$ | $O(1)$ |
| Asig Time | $O(1)$ | $O(1)$ | $O(1)$ |
| Public Key Size | $O(1)$ | $O(1)$ | $O(T)$ |
| Aver Time | $O(T)$ | $O(T)$ | $O(T)$ |

and sequential aggregation as in [20] or [19], instead of simultaneous aggregation of multiple signatures as in [8].

# 3. DEFINITIONS

A $FssAgg$ scheme is a key-evolving sequential aggregate signature scheme. As messages are generated sequentially in time, sequential (incremental) aggregation of signatures on these messages are performed. Like sequential aggregate signatures, it has Key Generation, Aggregate Signing and Aggregate Verification algorithms. Like key-evolving schemes, it has its operation divided into interverals, each of which use a different (but related) secret key to sign messages. The public key is left unchanged throughout the lifetime of the scheme while a Key Update algorithm is used to evolve the secret keys.

We now briefly define a $FssAgg$ signature scheme as follows.

*Definition 1.* A $FssAgg$ scheme is made up of four algorithms:

> *FssAgg.Kg* - key generation algorithm which takes as input a security parameter $k$, the total number of intervals $T$ and returns a pair $(SK_1, PK)$ where $SK_1$ is the initial private key and $PK$ the public key.
>
> *FssAgg.Asig* - sign-and-aggregate algorithm which takes as input a private key, a message to be signed and a signature-so-far (an aggregate 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.
>
> *FssAgg.Aver* - aggregate verification algorithm, which, on input of: a putative aggregate signature, a set of allegedly signed messages and a public key, outputs a binary value indicating whether the aggregate is valid.
>
> *FssAgg.Upd* - key update algorithm which takes as input the private key for the current interval and returns a new private key for the next interval (provided that the current interval does not exceed $T-1$.)

Any aggregate signature produced with *FssAgg.Asig* must be accepted by *FssAgg.Aver*.

This definition applies to both $FssAgg^m$ and $FssAgg^1$ signature schemes. A $FssAgg^m$ scheme places no restriction on the number of messages to be signed in each interval. Therefore, the frequency of key update can be chosen based on the perceived level of compromise possibility. It can be based on time (e.g., every hour), volume of activity (e.g., every 10 data entries) or some combination thereof.

A $FssAgg^1$ scheme, as mentioned earlier, allows at most one message to be signed in each interval. Ma and Tsudik [21] indeed define a $FssAgg^1$ signature scheme where key update is part of the sign-and-aggregate algorithm invoked immediately after each message is signed and aggregated.

The security of a *FssAgg* scheme is defined as the nonexistence of an adversary capable, within the rules of a certain game, of existentially forgery of a *FssAgg* signature, even in the event of the current secret key exposure. $FssAgg^m$ and $FssAgg^1$ signature schemes are examined under different games which capture the notions of existential unforgeability, forward security and aggregation security. We first define the forward-secure aggregate unforgeability of a $FssAgg^m$ signature scheme as follows.

*Definition 2.* A $FssAgg^m$ signature scheme is *forward-secure aggregate unforgeable* against adaptive chosen message attack *if no PPT adversary $\mathcal{A}$ can win the following game with non-negligible probability:*

1. *Setup.* The *FssAgg* forger $\mathcal{A}$ is given $PK$ and $T$.
2. *Queries.* The initial interval is $i = 1$. Proceeding adaptively, at interval $i$, $\mathcal{A}$ gets access to a signing oracle $\mathcal{O}_i$ under the current secret key $SK_i$. For each query, it also supplies a valid *FssAgg* signature $\sigma_{1,i-1}$ on messages $m_1, \cdots, m_{i-1}$ signed with secret keys $SK_1, \cdots, SK_{i-1}$ respectively, and an additional message $m_i$ to be signed by the oracle under key $SK_i$. $\mathcal{A}$ queries this as often as it wants until it indicates it is done for the current interval. Then $\mathcal{A}$ moves into the next interval $i+1$ and it is provided with a signing oracle $\mathcal{O}_{i+1}$ under the secret key $SK_{i+1}$. The query process repeats until $\mathcal{A}$ chooses to break in.
3. *Break-in.* At interval $b$, $\mathcal{A}$ chooses to break in and is given the break-in privilege, the secret key $SK_b$ for interval $b$.
4. *Response.* Finally, $\mathcal{A}$ outputs a *FssAgg* signature $\sigma_{1,t}$ on messages $m_1, \cdots, m_t$ under keys $SK_1, \cdots, SK_t$.

   The forger wins if (1) the *FssAgg* signature $\sigma_{1,t}$ is a valid *FssAgg* signature on messages $m_1, \cdots, m_t$ under keys $SK_1, \cdots, SK_t$, and (2) $\sigma_{1,t}$ is nontrivial, i.e., there exist at least one interval $k \in [1, t]$ s. t. $k \leq b$ during which $\mathcal{A}$ did not ask $\mathcal{O}_k$ for a signature query on message $m_k$ in the Query phase. The probability is over the coin tosses of the key-generation algorithm and of $\mathcal{A}$.

The security of a $FssAgg^1$ scheme can be defined the same way as that of a $FssAgg^m$ scheme, except that $\mathcal{A}$ is restricted to submitting only one signature query on a message of its choice for each interval in the Query phase.

*Definition 3.* A $FssAgg^1$ scheme is forward-secure aggregate unforgeable against adaptive chosen message attack *if no PPT adversary $\mathcal{A}$ can win the following game with non-negligible probability.*

1. *Setup.* The same as in Definition 2.
2. *Queries.* The initial interval is $i = 1$. Proceeding adaptively, at interval $i$, $\mathcal{A}$ gets access to a signing oracle $\mathcal{O}_i$ under the current secret key $SK_i$. For each query, it also supplies a valid *FssAgg* signature $\sigma_{1,i-1}$ on messages $m_1, \cdots, m_{i-1}$ signed with secret keys $SK_1, \cdots, SK_{i-1}$ respectively, and an additional message $m_i$ of its choice to be signed by the oracle under key $SK_i$. $\mathcal{A}$ queries this once and then moves into the next interval $i+1$ and it is provided with a signing oracle $\mathcal{O}_{i+1}$ under the secret key $SK_{i+1}$. The query process repeats until $\mathcal{A}$ chooses to break in.
3. *Break-in.* Same as in Definition 2.
4. *Response.* Same as in Definition 2.

   The forger wins if (1) the *FssAgg* signature $\sigma_{1,t}$ is a valid *FssAgg* signature on messages $m_1, \cdots, m_t$ under keys $SK_1, \cdots, SK_t$, and (2) $\sigma_{1,t}$ is nontrivial, i.e., there exist at least one interval $k \in [1, t]$ s. t. $k < b$ in which $\mathcal{A}$ did not ask $\mathcal{O}_k$ for signature query on message $m_k$ in the Query phase. The probability is over the coin tosses of the key-generation algorithm and of $\mathcal{A}$.

The security of a $FssAgg$ scheme subsumes security against truncation or deletion attacks. An adversary who compromises a signer has only one choice to produce a new valid aggregate: it includes the intact aggregate-so-far signature in future aggregated signatures. Therefore, a $FssAgg$ signature is append-only. It is computationally hard to selectively delete components of an already-generated aggregate signature.

# 4. QUALITY OF FORWARD SECURITY

We now discuss *Quality of Forward Security (QFS)*, a property of forward secure schemes. It determines to what extent forward security can be provided by a forward secure scheme. In our discussion, we assume that an interval $i$ starts at time $t_i^s$ and ends at time $t_i^e$. When the adversary compromises a system at interval $b$, it actually breaks in at a time $t_b$, such that $t_b \in [t_b^s, t_b^e)$.

## 4.1 QFS Levels

Forward security techniques are used to mitigate the effect of potential key compromise. QFS provided by a forward secure signature scheme can be evaluated by the number of messages whose authenticity remain trustworthy even after the signer is compromised, assuming the signer generates signatures as well as updates its secret signing key at fixed rates. We observe there are two levels of forward security: *interval-level* and *message-level*. The former guarantees authenticity of messages generated before compromise, i.e., prior to $t_b^s$, and the latter guarantees authenticity of messages generated before compromise time $t_b$ of interval $b$. Since $t_b > t_b^s$, a *message-level* scheme provides better forward security than an interval-level scheme.

Apparently, ordinary (non-aggregate) forward secure authentication schemes provide *interval-level* forward security. That is, once the secret key of interval $b$ is exposed, signatures generated in previous intervals are still trustworthy, while the signatures generated during and after $b$ are no longer to be trusted.

The BLS-$FssAgg$ scheme [21] provides *interval-level* forward security. It uses modular multiplication as the public aggregation function. As a consequence, it is hard to remove a component signature from the aggregate when the component signature is not known while it is easy to remove a component signature from the aggregate when the component signature is known. Once the attacker breaks in and obtains the current signing key, it can add new tags to the aggregate. It can also remove existing tags (from the break-in interval) from the aggregate: it simply re-generates those tags with the current key and removes them through modular division. Therefore, once the key is compromised, messages generated in the break-in interval (and future intervals) can not be trusted. Thus, $FssAgg$ schemes with reversible aggregate functions provide *interval-level* forward security.

In contrast, the *FssAgg* MAC scheme [21] provides *message-level* forward security. It uses a one-way hash function as its public aggregation function. Suppose at time $t_b^s + \Delta$ $(0 < \Delta < 1)$ of interval $b$, an attacker breaks in. All messages generated before time $t_b^s + \Delta$ are secure. Even is the adversary knows the current secret key, it can not change any part of the message body as doing so requires reversing the aggregation process. Therefore, it can not modify or delete any message for which the authentication tag has been already factored into the aggregate. $FssAgg$ schemes with irreversible aggregate functions provide *message-level* forward security. A $FssAgg^1$ scheme can be viewed as a special $FssAgg$ scheme which provides message-level forward security.

## 4.2 $FssAgg^m$ vs. $FssAgg^1$

When working with a *FssAgg* authentication scheme with *message level* forward security, a signer should sign and aggregate a message as soon as the message is generated to achieve the *message* level forward security.

When working with a $FssAgg^m$ authentication scheme with *interval* level forward security, a signer has two strategies to deal with messages generated in the same interval. The first strategy is that the signer signs a message as soon as the message is generated. The second strategy is that the signer waits until it is time to move into next interval and signs all messages generated during the current interval as a whole, e.g, using the $FssAgg^m$ scheme in a $FssAgg^1$ way.

These two strategies have different security implications under two different system models. In the first system model the "break-in" of a system equals to the "obtaining" of system information including the system's secret signing key. Most applications of $FssAgg$ authentication schemes use this model. For example, in the unattended sensor scenario, there is no way for a sensor to store its secret key separately from its data in a hope to make it harder for an attacker to access its key. An attacker who captures the sensor has control over both its data and its secret signing key. Even for a more powerful logging server, since its keys must be actively used for automatic data generation, it is hard to prevent attackers from gaining access to its keys [14]. Within

this system model, there is no difference between the use of these two strategies in term of security guarantee. However in term of performance, the second strategy is better than the first strategy since it involves less computation: the signer only signs ONE big message in each interval. Therefore in this model we can choose to use a $FssAgg^1$ scheme or use an $FssAgg^m$ scheme in the $FssAgg^1$ way to achieve better performance. In other words, we indeed require only $FssAgg^1$ signature scheme under this system model.

In the second system model, the "breaking-in" of the system does not mean the "obtaining" of the system's secret signing key. For example, the secret signing key might be encrypted under a second factor such as a human password [10, 18]. The secret key can be updated in the encrypted form without the presence of the human password. The human password is only required to sign a message. Without knowing the human password, the "breaking-in" attacker who obtains the secret signing key (in encrypted form) is not able to generate any authentication tag under this key. Therefore, it is not able to modify, delete existing data entries as well as insert new data entries. Within this system model, if the second strategy is used, a "breaking-in " attacker is able to modify messages whose authentication tags have not been folded into the aggregate. So only the first strategy should be used. In this case, we can choose to use a $FssAgg^m$ signature scheme instead of a $FssAgg^1$ signature scheme because the former supports more messages if they are initiated with the same maximum number of intervals $T$.

The encrypted key update mechanism in [10,18] is useful. However because of its requirement of human intervention, it cannot be used with *unattended* sensors deployed in remote places. It also cannot be used in data intensive applications, such as secure logging systems, where automatic (non-user intervention) data generation is required. In this paper, we focus on constructing $FssAgg^1$ schemes to be used in the first system model which requires only $FssAgg^1$ schemes. In this model, we can treat multiple message entries generated in the same interval as one BIG message and then let the signer sign only on this BIG message.

QFS shows that any general scenario using a $FssAgg^m$ scheme can equivalently use an $FssAgg^1$ scheme. It implies a $FssAgg^1$ is applicable in broad range of applications despite its restriction on the number of messages to be signed per interval.

# 5. BM-$FSSAGG^1$ SIGNATURE SCHEME

In this section, we give the construction of a $FssAgg^1$ signature scheme extended from the BM forward secure signature scheme [5]. We begin with the general idea behind this construction.

## 5.1 General Idea

The BM forward secure signature ($FSS$) scheme [5] is based on modifying the Fiat-Shamir ordinary signature scheme [11] and on the fact that squaring in $QR_n$ is a one-way permutation. Specifically, the public key contains $l$ values $v_1, \cdots, v_l \in QR_n$. The secret key for interval $j$ contains the root $s_{i,j} \in QR_n$ of $v_i$ of degree $2^{T-j+1}$ for $i = 1, \cdots, l$. Hence, Key Update is just squaring: $s_{i,j+1} \leftarrow s_{i,j}^2$. A signature in the $j$-th interval is a non-interactive proof of knowledge of $2^{T-j+1}$-th root in composite order group. An interactive zero-knowledge proof scheme is a $\sum$ protocol which has the following three-round communication structure:

1. Prover $\rightarrow$ Verifier: commitment $y$;
2. Prover $\leftarrow$ Verifier: challenge $c$;
3. Prover $\rightarrow$ Verifier: response $z$.

In the corresponding proof system of the BM $FSS$ scheme, at time $j$, the prover first selects a random number $r$, computes the commitment $y$ as $y \leftarrow r^{2^{T+j-1}}$ and sends $y$ to the verifier. After receiving the commitment, the verifier picks a random $l$-bits challenge $c_1 \cdots c_l$ and sends it to the prover. The prover calculates a response $z \leftarrow r \cdot \prod_{i=1}^{l} s_{i,j}^{c_i}$ under the challenge $c_1 \cdots c_l$ using the secret knowledge $s_{i,j}$ it has. The random $r$ is used to blind the response. The randomized response $z$ is sent to the verifier. Finally the verifier verifies whether the response is a valid answer under its challenge. The above protocol can be turned into a signature scheme using the Fiat-Shamir transform where the random challenge $c$ picked by the verifier is replaced as the hash output over the commitment $y$ and the to-be-signed message $m$, e.g. $c_1 \cdots c_l \leftarrow H(j, y, m)$.

A signature from the BM $FSS$ scheme has a format of $(z, c)$. Our goal is to find a way to aggregate these forward secure signatures $(z_1, c_1), \cdots, (z_t, c_t)$, one from each interval. At the same time, we want to make sure that aggregation should not violate the forward security of the BM $FSS$ scheme. Working on this signature format, the first challenge is how to aggregate the $c$ parts in these signatures. It seems it is simply impossible to aggregate them. Firstly, all the $c$ parts in signatures $(z_1, c_1), \cdots, (z_t, c_t)$ have to be presented individually to allow re-computation of the commitments in the verification process. Also we have no way to evolve these $c$ values because they appear mutually random - they are values from hash outputs. Therefore we do not know how to aggregate signatures in this form. Instead, we write a signature in the form of $(z, y)$. These two signature forms are equivalent as proofs of knowledge. The difference is that the former has better bandwidth than the latter as $c$ is a hash output with practical length of 160 bits while $y$ is usually an element of $Z_n^*$ with $|n| = 1024$.

In the BM $FSS$ scheme, the blind factor $r$ should be picked randomly so that the corresponding commitment $y$ appears randomly. However we cannot aggregate random $y$ values too. Therefore we want to find a way to evolve the blind factors in a way such that their corresponding commitments are related and can be computed from one another. Recall that the random value $r_j$ in interval $j$ is the $2^{T+j-1}$-th root of the commitment $y_j$ and the fact that squaring is a one-way permutation in $QR_n$. We can select $r_0$ randomly and set a common commitment $y$ as $y \leftarrow r_0^{2^{T+1}}$. Then we can evolve $r$s through squaring: $r_{j+1} \leftarrow r_j^2$. Our intuition of security here is that as long as $r_0$ is selected randomly, without knowing $r_0$ and any other intermediate values, $r_j$ appears random too. That is, an adversary should not be able to distinguish between a response blinded with $r_j$ and a response blinded with a random value $r$. Now $y$ is fixed and not a secret. We can move it as part of the public key. In fact, the way to set up $r_0$, $y$ and to evolve $r_j$ is exactly the same way which the BM $Fss$ scheme uses to set up the public/secret keys and evolve the secret key.

Now we only need to consider how to aggregate $z$s. As $r_j$ and $s_{i,j}$ are all squares now, $z_j$ is a square in $QR_n$ too.

$QR_n$ is a multiplicative group. Therefore we are able to use multiplication, the group operation over $QR_n$, to aggregate $z_j$s. The fact that multiplication is a group operation over $QR_n$ is very important for uniqueness of signatures.

## 5.2 The Scheme

After explaining the idea on how we aggregate forward secure BM signatures, we are ready to present our BM-$FssAgg^1$ signature scheme in Table 2.

The correctness of the BM-$FssAgg^1$ signature scheme is straightforward. For an aggregate signature $\langle t, \sigma_{1,t}\rangle$ over messages $m_1, \cdots, m_t$, we have $\sigma_{1,t} = \prod_{j=1}^{t} z_j$ where $z_j = r_j \cdot \prod_{i=1}^{l} s_{i,j}^{c_{i,j}} \bmod n$. In the first iteration of the *for* loop in the verification algorithm, we have:

$$\begin{aligned}
\sigma' &= \sigma_{1,t}^{2^{T+1-t}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t}} \\
&= (\prod_{j=1}^{t-1} z_j)^{2^{T+1-t}} \cdot z_t^{2^{T+1-t}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t}} \\
&= (\prod_{j=1}^{t-1} z_j)^{2^{T+1-t}} \cdot (r_t \cdot \prod_{i=1}^{l} s_{i,t}^{c_{i,t}})^{2^{T+1-t}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t}} \\
&= (\prod_{j=1}^{t-1} z_j)^{2^{T+1-t}} \cdot y^{-1} \cdot \prod_{i=1}^{l} u_i^{-c_{i,t}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t}} \\
&= (\prod_{j=1}^{t-1} z_j)^{2^{T+1-t}}
\end{aligned}$$

Therefore after the first iteration, $z_t$ is peeled off from $\sigma_{1,t}$ and all other $z_j$s are raised to the power of $2^{T+1-t}$. We set $\sigma' = (\prod_{j=1}^{t-1} z_j)^{2^{T+1-t}}$. In the second iteration, we have:

$$\begin{aligned}
&\sigma'^2 \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t-1}} \\
&= (\prod_{j=1}^{t-1} z_j)^{2^{T+2-t}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t-1}} \\
&= (\prod_{j=1}^{t-2} z_j)^{2^{T+2-t}} \cdot z_{t-1}^{2^{T+1-(t-1)}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,t-1}} \\
&= (\prod_{j=1}^{t-2} z_j)^{2^{T+2-t}}
\end{aligned}$$

So $z_{t-1}$ is peeled off and now all other $z_j$s are raised to a power of $2^{T+2-t}$. Therefore in the $j$-th iteration (for $j$ from $t$ to 1 ), $z_{t+1-j}$ is peeled off from the aggregate. In the last iteration, $\sigma'^2 \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,1}} = z_1^{2^T} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,1}} = 1$.

Our BM-$FssAgg^1$ signature scheme looks like a standard aggregate signature scheme from the view of the signer: it can aggregate signatures without any order simultaneously. However it acts as a sequential aggregate signature in the view of a verifier: the verifier must peel off component signatures from the aggregate from the outmost layer to the inner most layer in the verification process. It is inherently message order-preserving. Therefore, we do not need to add an index number to each message to preserve its order as the way in the BLS-$FssAgg$ signature scheme.

## 5.3 Security

Our BM-$FssAgg$ signature scheme is proven secure assuming the intractability of factoring of a large Blum-Williams integer. We state its security in the following theorem.

THEOREM 1. *Let $\mathcal{A}$ be an adversary that produces an existential forgery, in the forward secure aggregate security attack model defined in Definition 3, against the BM-$FssAgg^1$ signature scheme instantiated for $T$ intervals. Assume that $\mathcal{A}$ makes no more than $q_s$ signature queries, and succeeds with probability $\epsilon$ in time $\tau$. Then there exists an algorithm $\mathcal{B}$ which factors a Blum-Williams integer into its two prime factors in time $\overline{\tau} = \tau + O(q_s)$ with success probability $\overline{\epsilon} \geq \epsilon/T$.*

Please see Appendix A for proof.

# 6. AR-$FSSAGG^1$ SIGNATURE SCHEME AND OTHERS

We can apply the same idea described in Section 5.1 onto the AR $FSS$ [3] to get the AR-$FssAgg^1$ signature scheme. We present the AR-$FssAgg^1$ signature scheme in Table 3 and omit further discussion for the sake of page limitation. Its correctness can be proved in the similar way as we prove the correctness of the BM-$FssAgg^1$ signature scheme.

Like the BM-$FssAgg^1$ signature scheme, the AR-$FssAgg^1$ signature scheme is also proven secure assuming the intractability of factoring of a large Blum-Williams integer. We state its security in the following theorem.

THEOREM 2. *Let $\mathcal{A}$ be an adversary that produces an existential forgery, in the forward secure aggregate security attack model defined in Definition 3, against the AR-$FssAgg^1$ signature scheme instantiated for $T$ intervals. Assume that $\mathcal{A}$ makes no more than $q_s$ signature queries, and succeeds with probability $\epsilon$ in time $\tau$. Then there exists an algorithm $\mathcal{B}$ which factors a Blum-Williams integer into its two prime factors in time $\overline{\tau} = \tau + O(q_s)$ with success probability $\overline{\epsilon} \geq \epsilon/T$.*

Please see Appendix B for proof.

The idea described in Section 5.1 can also be applied onto the GQ forward-secure signature scheme [15]. However, due to its expensive key update ($O(T)$), the resulted $FssAgg^1$ scheme has a very expensive *signer computation per message* cost [1] in the order of $O(T)$ which makes it non-practical. On the other hand, the BM- and AR-$FssAgg^1$ schemes have $O(1)$ *signer computation per message* cost.

# 7. AUDIT LOG SECURITY

System logs are an important part of any secure IT system. They record noteworthy events, such as user activity, program execution steps, system resource usage as well as system state and important data modifications. They also provide a valuable view of the past and current states of complex systems. 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 its presence in order to hide attack details or frame innocent users. Therefore, providing *forward-secure stream integrity* - resistance against post-compromise insertion, alteration,

[1] *Signer computation per message* cost includes the cost of one aggregate and sign operation plus the cost of one key update operation.

**Table 2: The BM-$FssAgg^1$ signature scheme**

| | |
|---|---|
| $\mathsf{KeyGen}(k, l, T)$<br><br>Generate random distinct $k/2$-bit primes $p$, $q$, each congruent to 3 mod 4<br>$n \leftarrow pq$<br>for $i = 1, \cdots, l$ pick $s_{i,0} \stackrel{R}{\leftarrow} \mathbb{Z}_n^*$ and compute $u_i \leftarrow 1/s_{i,0}^{2^{T+1}} \bmod n$<br>pick $r_0 \stackrel{R}{\leftarrow} \mathbb{Z}_n^*$ and compute $y \leftarrow 1/r_0^{2^{T+1}} \bmod n$<br>$SK_1 \leftarrow (n, T, 1, s_{1,0}^2, s_{2,0}^2, \cdots, s_{l,0}^2, r_0^2)$<br>and $PK \leftarrow (n, T, u_1, \cdots, u_l, y)$<br>return $(SK_1, PK)$ | $\mathsf{Upd}(SK_{t-1})$ $(t = 2, \cdots, T)$<br><br>Let $SK_{t-1} = (n, T, t-1, s_{1,t-1}, s_{2,t-1}, \cdots, s_{l,t-1}, r_{t-1})$<br>return $SK_t = (n, T, t, s_{1,t-1}^2, s_{2,t-1}^2, \cdots, s_{l,t-1}^2, r_{t-1}^2)$ |
| $\mathsf{AggSign}(SK_t, M, \sigma_{1,t-1})$ $(t = 1, \cdots, T$ and $\sigma_{1,0} = 1)$<br><br>Let $SK_t = (n, T, t, s_{1,t}, s_{2,t}, \cdots, s_{l,t}, r_t)$<br>$c_1 \cdots c_l \leftarrow H(t, y, M)$<br>$z_t \leftarrow r_t \cdot \prod_{i=1}^{l} s_{i,t}^{c_i} \bmod n$<br>$\sigma_{1,t} \leftarrow \sigma_{1,t-1} \cdot z_t \bmod n$<br>return $\langle t, \sigma_{1,t} \rangle$ | $\mathsf{AggVerify}(PK, M_1, \cdots, M_t, \langle t, \sigma_{1,t} \rangle)$<br><br>Let $PK = (n, T, u_1, \cdots, u_l, y)$<br>for $j = t \cdots 1$,<br>• compute $c_{1,j} \cdots c_{l,j} \leftarrow H(j, y, M_j)$<br>• if $j = t$, compute $\sigma' \leftarrow \sigma_{1,t}^{2^{T+1-t}} \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,j}}$<br>• else $\sigma' \leftarrow \sigma'^2 \cdot y \cdot \prod_{i=1}^{l} u_i^{c_{i,j}}$<br>if $\sigma' = 1$ then return 1 else return 0 |

deletion and re-ordering of log entries – is critical for any application usin secure logging.

A number of cryptographic approaches have been proposed to address audit log integrity for logs generated and stored on local logging servers [6, 7, 13, 23]. Bellare and Yee were the first to define the *forward-secure stream integrity*, a property required for any secure audit log system [6,7]. They proposed using forward-secure MACs to mitigate potential log server compromise and indexing to preserve log entry order. Schneier and Kelsey constructed a similar system, also using forward-secure MACs [23]. Instead of indexing log entries, they employed a one-way hash chain to link log entries and preserver ordering. Holt extended [23] to the public key setting [13]. Unfortunately, none of these schemes defend against *truncation attack* - a special kind of deletion attack, whereby the attacker deletes a contiguous subset of tail-end log entries. The truncation attack seems to be quite fundamental and it is surprising that it has not been addressed thus far. Also all prior schemes are inefficient in terms of both storage and communication costs. Also, symmetric-key-based schemes (Schneier-Kelsey and Bellare-Yee) suffer from *delayed detection attacks*, since they rely on a trusted server to aid users in verifying the audit log integrity. Modifications can not be detected until the entire log is uploaded to the trusted server.

We claim that $FssAgg$ authentication implies *forward-secure stream integrity* due to its forward secure aggregate unforgeability, i.e.:

> *Forward Security:* In a $FssAgg$ scheme, the secret signing key is updated via a one-way function. An attacker is thus unable to recover previous keys from the compromised key and unable to forge signatures from prior intervals.
>
> *Stream Security.* The sequential aggregation process in a $FssAgg$ scheme preserves message order thus providing stream security and making message re-ordering impossible.
>
> *Integrity.* Any insertion of new messages as well as modification or deletion of existing messages renders the final aggregate signature unverifiable.

Armed with the above, we can build a secure logging system from any $FssAgg$ signature scheme. Furthermore, the resulting secure logging system would inherit the provable security of the underlying $FssAgg$ signature scheme.

We can initialize the system with the log server generating a public/secret key-pair and registering its public key with a well-known Certification Authority (CA). An authenticated log file in our scheme would consist of two parts: log entries $[L_0, \cdots, L_t]$ and a single $FssAgg$ signature, $\sigma_{0,t}$. When a client program requests an audit log entry $L_i$ to be made, the log server first updates the $FssAgg$ signature with the aggregate-and-sign algorithm $Asig$ under its current secret key $SK_i$. It then evolves the private key through the $Upd$ algorithm and securely erases previous secret key. (Note that key update is invoked immediately after the aggregate signature is generated.) The log server closes the log file by creating a special closing message as the final entry $L_f$, updating the $FssAgg$ signature accordingly, and securely erasing its last secret key. Anyone who obtains a copy of the log file can get the public key from the CA and validate the file.

# 8. PERFORMANCE EVALUATION

We now analyze the performance of proposed $FssAgg^1$ schemes. We begin by accessing the cost in term of basic cryptographic operations and then discuss experimental results obtained from a prototype implementation.

## 8.1 Performance Evaluation

A $FssAgg$ signature scheme can be evaluated by the following parameters:

**Table 3: The AR-$FssAgg^1$ signature scheme**

| | |
|---|---|
| $\mathsf{Kg}(k, l, T)$<br><br>Generate random distinct $k/2$-bit primes $p$, $q$, such that:<br>$p \equiv q \equiv 3 \pmod 4$<br>$2^{k-1} \leq (p-1)(q-1)$<br>$pq < 2^k$<br>$n \leftarrow pq$<br>pick $s_0 \overset{R}{\leftarrow} \mathbb{Z}_n^*$ and compute $u \leftarrow 1/s_0^{2^{l(T+1)}} \mod n$<br>pick $r_0 \overset{R}{\leftarrow} \mathbb{Z}_n^*$ and compute $y \leftarrow 1/r_0^{2^{l(T+1)}} \mod n$<br><br>$SK_1 \leftarrow (n, T, 1, s_0^{2^l}, r_0^{2^l})$<br>$PK \leftarrow (n, T, u, y)$<br>return $(SK_1, PK)$ | $\mathsf{Upd}(SK_{t-1})$ $(t = 2, \cdots, T)$<br><br>Let $SK_{t-1} = (n, T, t-1, s_{t-1}, r_{t-1})$<br>return $SK_t = (n, T, t, s_{t-1}^{2^l}, r_{t-1}^{2^l})$ |
| $\mathsf{Asig}(SK_t, M, \sigma_{1,t-1})$ $(t = 1, \cdots, T$ and $\sigma_{1,0} = 1)$<br><br>Let $SK_t = (n, T, t, s_t, r_t)$<br>$c \leftarrow H(t, y, M)$<br>$z_t \leftarrow r_t \cdot s_t^c \mod n$<br>$\sigma_{1,t} \leftarrow \sigma_{1,t-1} \cdot z_t \mod n$<br>return $\langle t, (\sigma_{1,t}) \rangle$ | $\mathsf{Aver}(PK, M_1, \cdots, M_t, \langle t, (\sigma_{1,t}, y) \rangle$<br><br>Let $PK = (n, T, u, y)$<br>for $j = t \cdots 1$,<br>• compute $c_j \leftarrow H(j, y, M_j)$<br>• if $j = t$, compute $\sigma' \leftarrow \sigma_{1,t}^{2^{l(T+1-t)}} \cdot y \cdot u^{c_j}$<br>• else $\sigma' \leftarrow \sigma'^{2^l} \cdot y \cdot u_j^{c_j}$<br><br>if $\sigma' = 1$ then return 1 else return 0 |

1. size of aggregate signature
2. size of secret key
3. complexity of secret key update (on-line)
4. complexity of aggregate signing (on-line)
5. complexity of generation (off-line)
6. size of public key
7. complexity of aggregate verifying (on-line)

The first five parameters represent *signer efficiency* and the last two – *verifier efficiency*. The size of public and secret keys correspond to *space efficiency* and signing, verifying and key update complexity correspond to *time efficiency*. We view signer efficiency as being more important than verifier efficiency and space efficiency – more important than time efficiency. In envisaged applications, such as auditing and data gathering systems, signers are the monitoring or data gathering devices performing operations in real time and generating/collecting data at high rates. They might also have limited storage facilities and limited power (e.g., battery-operated sensors in unattended environments). A verifier, in contrast, is assumed to be a powerful entity (e.g, a server or a sink) that collects and verifies data generated/gathered by aforementioned devices. It can perform these tasks off-line.

We evaluate size parameters in number of elements in $Z_n^*$. We evaluate the time parameters in terms of basic cryptographic operations. $Sqt^t(m)$ denotes $t$ modular squarings and $Mult^t(m)$ denotes $t$ modular multiplication with modulus of size $m$. We estimate that one modular exponentiation $Exp_l^1(m)$ with exponent size of $l$ and modulus size $m$ roughly equals $l$ modular squarings and $l/2$ modular multiplications: $Exp_l^1(n) = Sqt^l(n) + Mult^{l/2}(n)$. In practice, modular squaring is much faster than modular multiplication. Evaluation results are shown in Table 8.1. Both schemes have constant-size parameters. Signer computation (key update and aggregate signature generation) is constant, while verifier computation (aggregate verification) requires $O(T)$ complexity.

AR-$FssAgg^1$ has smaller public and private keys; it requires only 2 units of storage for both signer and verifier. BM-$FssAgg^1$ requires $l + 1$ units of storage for both signer and verifier. BM-$FssAgg^1$ requires fewer squarings and multiplications in key update, signature generation and verification. It is thus more efficient in term of computation. We recommend its use in applications where computation and communication are the first two priorities.

## 8.2 Implementation

We implemented both proposed schemes on an Intel Dual-Core 1.73GHz laptop with 1GB RAM under Linux. We implemented BM-$FssAgg^1$ and AR-$FssAgg^1$ using Shoup's NTL library [1]. We fixed security parameters at $k = 1024$ and $l = 160$. For comparison, we also implemented the BLS-$FssAgg$ scheme using the Stanford PBC library [2] on the same hardware platform. We used a singular curve $Y^2 = X^3 + X$ defined on field $F_q$ for $|q| = 512$ and group order $|p| = 160$ where $p$ is a Solinas prime. Such groups have the fastest pairing operations [2]. We measured signer cost by signature generation and key update on a per message basis. We measured verifier cost over an aggregate signature $\sigma_{1,t}$ when $t = 100, 1,000$ and $10,000$ which corresponds to small, medium, and large data sets, respectively. The results are shown in Table 8.2.

Although key update in both schemes requires more com-

**Table 4: Costs in terms of cryptographic operations; $k = |n|$ and $l$ are security parameters and $t$ denotes the current interval.**

| Parameters | BM $FssAgg^1$ | AR $FssAgg^1$ |
|---|---|---|
| Signature Size | $1/O(1)$ | $1/O(1)$ |
| Secret Key Size | $l + 1/O(1)$ | $2/O(1)$ |
| Key Update Time | $Sqr^{l+1}(n)/O(1)$ | $Sqr^{2l}(n)/O(1)$ |
| Asig Time | $Mult^{1+l/2}(n)/O(1)$ | $Sqr^{l}(n) + Mult^{2+l/2}(n)/O(1)$ |
| Public Key Size | $l + 1/O(1)$ | $2/O(1)$ |
| Aver Time | $Sqr^{T}(n) + Mul^{(1+l/2)t}(n)/O(T)$ | $Sqr^{l(T+t)}(n) + Mult^{(2+l/2)t}(n)/O(T)$ |

**Table 5: Operation Timing in $msec$.**

| | | BM-$FssAgg^1$ | AR-$FssAgg^1$ | BLS-$FssAgg^1$ |
|---|---|---|---|---|
| Signer Cost (per msg) | *Asig* | 2.09 | 4.39 | 30 |
| | *Upd* | 3.46 | 7.27 | 0.002 |
| | total | 5.55 | 11.66 | 30.00 |
| Verifier Cost | $t = 100$ | 211.97 | 810.88 | $3.30 \times 10^3$ |
| | $t = 1000$ | $2.13 \times 10^3$ | $8.16 \times 10^3$ | $29.3 \times 10^3$ |
| | $t = 10000$ | $21.35 \times 10^3$ | $80.84 \times 10^3$ | $330.72 \times 10^3$ |

putation than key update in BLS-$FssAgg$ (where key update is merely a hash), aggregate signature generation in our schemes requires much less computation than that in BLS-$FssAgg$. Therefore, signer costs (per message) - one aggregation, one sign operation plus one key update - in our schemes are less than those in BLS-$FssAgg$. Specifically, signer computation per message in BM-$FssAgg^1$ and AR-$FssAgg^1$ is, respectively, 6 and 3 times faster than that in BLS-$FssAgg$. Verifier computation in both new schemes also costs less than that in BLS-$FssAgg$: 16 and 4 times faster, respectively.

In conclusion, new $FssAgg^1$ schemes perform much better in almost all parameters than the pre-existing BLS-$FssAgg$ scheme. Among the two new schemes, BM-$FssAgg^1$ is more efficient in computation, while AR-$FssAgg^1$ is more efficient in storage. They have the same communication efficiency.

# 9. DISCUSSION

*Cross User Aggregation and Verification.* How fast can aggregate verification be? Can we have constant verification time? So far, all aggregate signatures including ours require $O(T)$ aggregate verification time. (Although the aggregate verification in LOSSW [19] requires only two pairing operations regardless of how many intervals or signers, it do require $O(T)$ multiplications.) Note that the cost of the first iteration in both our aggregate verification functions dominates the total cost as it requires $O(T)$ operations. A possible way to reduce aggregate verification cost is to aggregate $FssAgg$ signatures from multiple signers first and then verify this super aggregate signature to amortize the cost in the first iteration among multiple users.

*From $FssAgg^1$ to $FssAgg^m$.* We can trivially extend our $FssAgg^1$ schemes to be $FssAgg^m$ schemes at a cost of increasing size of secret key and public key. That is, let $m$ be the maximum number of messages generated in each interval. In the key generation stage, we selects $m$ random numbers $r_{0,1}, \cdots, r_{0,m}$ and calculate $m$ corresponding common commitments $y_1, \cdots, y_m$ such that $y_i = r_{0,1}^{2^{T+1}}$. We use $y_i$ to generate the challenge when we sign the $i$-th message of each interval. Therefore by additional $m$ units of space, we can support up to $mT$ messages.

# 10. CONCLUSION

In this paper, we identified general scenarios that use a $FssAgg^m$ scheme can equivalently use a $FssAgg^1$ scheme because of QFS. We proposed two practical $FssAgg^1$ signature schemes that can be used in applications which require both forward security and storage/communication efficiency. Our schemes are extended from existing forward secure signature schemes and they are provable secure. They outperform the BLS-$FssAgg$ signature scheme in almost all parameters. Especially they have compact public key and lower aggregate verification complexity. From our implementation, the BM-$FssAgg^1$ is 16 times faster than the BLS-$FssAgg$ scheme while the AR-$FssAgg^1$ is 4 times faster than the BLS-$FssAgg$ scheme in aggregate verification. Therefore our schemes are more practical than the BLS-$FssAgg$ signature scheme to be used in applications with either weak unattended device such as sensors or with intensive data generation such as secure logging. We applied them to secure logging systems to provide *forward secure stream integrity* for audit logs and evaluated their viability with implementation. However, our construction is still trivial. We will investigate non-trial $FssAgg$ signature scheme construction in the future.

## Acknowledgments

The author thanks Professor Gene Tsudik, her PHD advisor, for his kind support help on the final version of this paper. The author also thanks Dr. Guilin Wang for his time analyzing the security of proposed schemes. Furthermore, the author is grateful to the anonymous reviewers for their valuable comments.

# 11. REFERENCES

[1] Ntl: a library for doing number theory. *http://www.shoup.net/ntl/*.

[2] Pbc library benchmarks. *http://crypto.stanford.edu/pbc/times.html*.

[3] M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. In *Asiacrypt 2000*, pages 116–129, 2000.

[4] R. Anderson. Two remarks on public-key cryptology - invited lecture. In *Fourth ACM Conference on Computer and Communications Security (CCS)*, April 1997.

[5] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In *Proc. of Advances in Cryptology - Crypto 99*, pages 431–448, August 1999.

[6] M. Bellare and B. Yee. Forward integrity for secure audit logs. In *Technical Report, Computer Science and Engineering Department, University of San Diego*, November 1997.

[7] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Proc. of CT-RSA'03*, 2003.

[8] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proc. of Eurocrypt 2003*, pages 416–432, May 2003.

[9] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Asiacrypt*, volume 2248 of *Lecture Notes in Computer Science*, 2001.

[10] X. Boyen, H. Shacham, E. Shen, and B. Waters. Forward-secure signatures with untrusted update. In *ACM CCS'06*, October 2006.

[11] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and siganture problems. In *Advances in Cryptology - CRYPTO'86*, pages 186–194, August 1986.

[12] C. G. Günther. An identity-based key-exchange protocol. In *EUROCRYPT '89: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 29–37, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[13] 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*, pages 203–211, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[14] G. Itkis. Forward security (adaptive cryptography: time evolution). *Handbook of Information Security*, 2006.

[15] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 332–354, London, UK, 2001. Springer-Verlag.

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

[17] H. Krawczyk. Simple forwardsecure signatures from any signature scheme. In *Proc. 7th ACM Conference on Computer and Communication Security (CCS)*, pages 108–115, November 2000.

[18] B. Libert, J. J. Quisquater, and M. Yung. Forward-secure signatures in untrusted update environments: efficient and generic constructions. In *ACM CCS'07*, Oct 2007.

[19] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters. Sequential aggregate signatures and multisignatures without random oracles. In *Prof. of Eurocrypt 2006*, May 2006.

[20] A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham. Sequential aggregate signatures from trapdoor permutations. In *Proc. of Eurocrypt 2004*, pages 245–254, November 2001.

[21] D. Ma and G. Tsudik. Forward-secure sequentical aggregate authentication. In *Proceedings of IEEE Symposium on Security and Privacy 2007*, May 2007.

[22] T. Malkin, D. Micciancio, and S. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Proc. of Eurocrypt (Eurocrypt'02)*, 2002.

[23] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

[24] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, pages 159–176, 1999.

# APPENDIX

## A. PROOF OF THEOREM 1

We will use the following Lemma [5] in our proof:

LEMMA 1. *Suppose $n$ is a Blum-Williams integer. Suppose $a, a_1, \cdots, a_t \in Z_n^*$ and $a$ is a square modulo $n$. Suppose $m, m_1, \cdots, m_t$ are integers such that $m_1, \cdots, m_t > m \geq 0$. Suppose*

$$a^{2^m} = \prod_{j=1}^{t} a_j^{2^{m_j}} \ mod\ n.$$

*Then*

$$a = \prod_{j=1}^{t} a_j^{2^{m_j - m}} \ mod\ n.$$

PROOF. Suppose there exist a forger $\mathcal{A}$ against the BM-$FssAgg^1$ signature scheme that succeeds with $\epsilon$ in time $\tau$. We construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ as a subroutine to factor a given Blum-Williams integer $n$ with a probability of a lower bound at $\epsilon/T$ within time $\sim \tau$.

$\mathcal{B}$ is given a Blum-Williams integer $n$. Its goal is to output a pair $(p, q)$ such that $n = pq$. $\mathcal{B}$ interacts with $\mathcal{A}$ as follows.

[**Setup**] $\mathcal{B}$ picks $x_j \stackrel{R}{\leftarrow} \mathbb{Z}_n^*$ for $j = 1, \cdots, l$ and $w \stackrel{R}{\leftarrow} \mathbb{Z}_n^*$. Then it chooses a time period $b'$ when to set a "trap". Now it sets $s_{j,b'} \leftarrow x_j^2$ and $r_{b'} \leftarrow w^2$. It computes $u_j \leftarrow s_{j,b'}^{2^{T+1-b'}}$ and $y \leftarrow r_{b'}^{2^{T+1-b'}}$. It gives $\mathcal{A}$ the public key $PK = \{n, T, u_1, \cdots, u_l, y\}$.

[**Queries**] Starting from time period 1. At time period $i$, $\mathcal{A}$ requests a BM-$FssAgg^1$ signature on a message $m_i$ of her choice under the public key $PK$. She also supplies $\mathcal{B}$ with an aggregate signature $\sigma_{1,i-1}$ on messages $m_1, \cdots, m_{i-1}$ (we assume $\sigma_{1,0} = 1$ on any arbitrary message). $\mathcal{B}$ maintains a message-signature table with a tuple form of $\langle m^{(j)}, \sigma^{(j)}, \sigma^{(1,j)} \rangle$ as explained below. $\mathcal{B}$ responds to this query using the following steps:

1 Algorithm $\mathcal{B}$ first checks the validity of $\sigma_{1,i-1}$ using $PK$. If the signature cannot be verified, $\mathcal{B}$ aborts, fails in factoring. Otherwise, it goes to step 2.

2 For $j = 1$ to $i-1$, $\mathcal{B}$ checks whether $m_j$ appears in the $j$-th entry of its message-signature table, e.g. whether $m_j = m^{(j)}$. If there exist an index $k$ such that $m_k \neq m^{(k)}$, and if $k < b'$, then $\sigma_{1,i-1}$ is a valid forgery, algorithm $\mathcal{B}$ starts to factor $n$ as described below. Otherwise, it goes to step 3.

3 If $i \geq b'$, $\mathcal{B}$ computes $\sigma_i = r_i \prod_{j=1}^{l} s_{j,i}^{c_j}$ ($\mathcal{B}$ knows the secret keys in these time periods) where $(c_1 \cdots c_l) = H(i, y, m_i)$ and $\sigma_{1,i} = \sigma_{1,i-1} \cdot \sigma_i$. If $i < b'$, $\mathcal{B}$ goes to its signing oracle to get the corresponding BM-$FssAgg^1$ signature $\sigma_{1,i}$ and computes $\sigma_i = \sigma_{1,i}/\sigma^{(1,i-1)}$. $\mathcal{B}$ inserts the tuple $(m_i, \sigma_i, \sigma_{1,i})$ as the $i$-th entry into its message-signature table. $\mathcal{B}$ replies $\sigma_{1,i}$ to $\mathcal{A}$.

[**Break-in.**] When $\mathcal{A}$ decides to break in and query the secret information for some $b$-th period, if $b \leq b'$ then it aborts its run (i.e., in this case $\mathcal{B}$ fails to factor); if $b > b'$ then $\mathcal{B}$ provides $\mathcal{A}$ the secret information for that period ($\mathcal{B}$ knows it).

[**Response.**] Finally $\mathcal{A}$ outputs a forgery $\sigma_{1,i}$ ($i \leq T$) over messages $m_1, \cdots, m_i$. That $\sigma_{1,i}$ is a valid BM-$FssAgg^1$ forgery means that: (1) it is a valid signature, e.g., it can be verified with $PK$; and 2) it is nontrivial: there exist AT LEAST one index $k$ such that $1 \leq k < b'$ and $m_k \neq m(k)$ in $\mathcal{B}$'s message-signature table. Let $\kappa$ denote the set of such indexes. We have $|\kappa| \in [1, b'-1]$. If $\sigma_{1,i}$ is not a valid forgery, $\mathcal{B}$ aborts and fails in factoring. Otherwise, $\mathcal{B}$ computes $\sigma' = \sigma_{1,i}/\prod_{k\in\kappa} \sigma_k$. In the following, we only show how $\mathcal{B}$ proceeds to factor $n$ in the case where $|\kappa| = 2$, e.g., there exist two indexes $k_1$ and $k_2$ such that $1 \leq k_1 < k_2 < b'$, $m_{k_1} \neq m^{(k_1)}$, and $m_{k_2} \neq m^{(k_2)}$. $\mathcal{B}$ proceeds in the similar way in other cases when $|\kappa| = 1$ or $2 < |\kappa| \leq b'-1$.
$\mathcal{B}$ first computes $(c_{1,k_1} \cdots c_{l,k_1}) = H(k_1, y, m_{k_1})$ and $(c_{1,k_2} \cdots c_{l,k_2}) = H(k_2, y, m_{k_2})$. Let

$$A = r_{b'} \cdot \prod_{j=1}^{l} s_{i,b'}^{c_{i,k_2}\cdot(k_2-k_1)+c_{i,k_1}}$$

, and

$$a_1 = w \cdot \prod_{j=1}^{l} x_i^{c_{i,k_2}\cdot(k_2-k_1)+c_{i,k_1}}$$

. Then $a_1^2 = A$, e.g., $a_1$ is a square root of $A$. Now $\mathcal{B}$ computes $\sigma'' = 1/\sigma'$ ($\sigma'$ is a square in the multiplicative group $QR_n$ and its inverse can be computed in polynomial time using the Extended Euclid Algorithm). Finally $\mathcal{B}$ computes $a_2 = (\sigma'')^{2^{b'-k_1-1}}$ as another square root of A (because $b' > k_2 > k_1 \geq 1$, $b' - k_1 - 1 \geq 1$). If $a_2 \equiv \pm a_1$ then $\mathcal{B}$ fails to factor $n$; else it computes $h \leftarrow \gcd(a_1 - a_2, n)$ and outputs $(h, n/h)$ as factors of $n$.

Now we only need to prove that $a_2$ is a square root of A. Because $\sigma_{1,i}$ is a valid BM-$FssAgg^1$ signature, according to the verification process, we have the following equation:

$$[(\sigma')^{2^{T-k_2+1}} \cdot y \cdot \prod_{j=1}^{l} u_j^{c_{i,k_2}}]^{2^{k_2-k_1}} \cdot y \cdot \prod_{j=1}^{l} u_j^{c_{i,k_1}} = 1$$

That equals:

$$(\sigma'')^{2^{T-k_1+1}} = y^{k_2-k_1+1} \cdot \prod_{j=1}^{l} u_j^{c_{i,k_2}\cdot(k_2-k_1)+c_{i,k_1}}$$

Because $u_j = s_{j,b'}^{2^{T+1-b'}}$ and $y = r_{b'}^{2^{T+1-b'}}$, the above equation can be written as:

$$\begin{aligned}(\sigma'')^{2^{T-k_1+1}} &= (r \cdot \prod_{j=1}^{l} s_{i,b'}^{c_{i,k_2}\cdot(k_2-k_1)+c_{i,k_1}})^{2^{T+1-b'}} \\ &= A^{2^{T+1-b'}}\end{aligned} \quad (1)$$

Because $T-k_1+1 > T+1-b'$ and $A$ is a square, according to Lemma 1, we have

$$(\sigma'')^{2^{b'-k_1}} = A$$

Therefore $a_2^2 = ((\sigma'')^{2^{b'-k_1-1}})^2 = A$. That is, $a_2$ is a square root of $A$.

Algorithm $\mathcal{B}$ makes as many as signature queries as $\mathcal{A}$ makes. Algorithm $\mathcal{B}$'s running time is that of $\mathcal{A}$, plus the overhead in handling $\mathcal{A}$'s signature queries.

If $\mathcal{A}$ succeeds with probability of $\epsilon$ in forging, $\mathcal{B}$ succeeds at least with probability roughly $\epsilon/T$. The argument is outlined as follows. First, the view of $\mathcal{A}$ that $\mathcal{B}$ produces is computationally indistinguishable from the view of $\mathcal{A}$ interacting with a real $FssAgg^1$ signing oracle. Conditioned on $\mathcal{B}$ choosing the value of $b'$ as the period for which $\mathcal{A}$ eventually output a forgery, we have the probability that $\mathcal{B}$ outputs two factors is the same probability that $\mathcal{A}$ succeeds in forging, i.e., probability $\epsilon$. Since choosing the "right" $b'$ happens with probability $1/T$ we get that $\epsilon/T$ is an approximate lower bound on the forging probability of $\mathcal{B}$.

□

# B. PROOF OF THEOREM 2

Proof of Theorem 2 resembles a lot to the Proof of Theorem 1.

PROOF. Suppose there exist a forger $\mathcal{A}$ against the AR-$FssAgg^1$ signature scheme that succeeds with $\epsilon$ in time $\tau$. We construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ as a subroutine to factor a given Blum-Williams integer $n$ with a probability of a lower bound at $\epsilon/T$ within time $\sim \tau$.

$\mathcal{B}$ is given a Blum-Williams integer $n$. Its goal is to output a pair $(p, q)$ such that $n = pq$. $\mathcal{B}$ interacts with $\mathcal{A}$ as follows.

[**Setup**] $\mathcal{B}$ picks $x \overset{R}{\leftarrow} \mathbb{Z}_n^*$ and $w \overset{R}{\leftarrow} \mathbb{Z}_n^*$. Then it chooses a time period $b'$ when to set a "trap". Now it sets $s_{b'} \leftarrow x^{2^l}$ and $r_{b'} \leftarrow w^{2^l}$. It computes $u \leftarrow s_{b'}^{2^{l(T+1-b')}}$ and $y \leftarrow r_{b'}^{2^{l(T+1-b')}}$. It gives $\mathcal{A}$ the public key $PK = \{n, T, u, y\}$.

[**Queries**] The same as in Proof of Theorem 1.

[**Break-in.**] The same as in Proof of Theorem 1.

[**Response.**] Finally $\mathcal{A}$ outputs a forgery $\sigma_{1,i}$ ($i \leq T$) over messages $m_1, \cdots, m_i$. That $\sigma_{1,i}$ is a valid AR-$FssAgg^1$ forgery means that: (1) it is a valid signature, e.g., it can be verified with $PK$; and 2) it is nontrivial: there exist AT LEAST one index $k$ such that $1 \leq k < b'$ and $m_k \neq m(k)$ in $\mathcal{B}$'s message-signature table. Let $\kappa$ denote the set of such indexes. We have $|\kappa| \in [1, b'-1]$.

If $\sigma_{1,i}$ is not a valid forgery, $\mathcal{B}$ aborts and fails in factoring. Otherwise, $\mathcal{B}$ computes $\sigma' = \sigma_{1,i} / \prod_{k \in \kappa} \sigma_k$. In the following, we only show how $\mathcal{B}$ proceeds to factor $n$ in the case where $|\kappa| = 2$, e.g., there exist two indexes $k_1$ and $k_2$ such that $1 \leq k_1 < k_2 < b'$, $m_{k_1} \neq m^{(k_1)}$, and $m_{k_2} \neq m^{(k_2)}$. $\mathcal{B}$ proceeds in the similar way in other cases when $|\kappa| = 1$ or $2 < |\kappa| \leq b' - 1$.
$\mathcal{B}$ computes $c_{k_1} = H(k_1, y, m_{k_1})$, $c_{k_2} = H(k_2, y, m_{k_2})$. Let

$$A = r_{b'}^{2^{l(k_2-k_1)+1}} \cdot s_{b'}^{2^{l(k_2-k_1)} c_{k_2} + c_{k_1}}$$

, and

$$a_1 = w^{2^{l(k_2-k_1)+1}} \cdot x^{2^{l(k_2-k_1)} c_{k_2} + c_{k_1}}$$

. Then $a_1^2 = A$, e.g., $a_1$ is a square root of $A$. Now $\mathcal{B}$ computes $\sigma'' = 1/\sigma'$ ($\sigma'$ is a square in the multiplicative group $QR_n$ and its inverse can be computed in polynomial time using the Extended Euclid Algorithm). Finally $\mathcal{B}$ computes $a_2 = (\sigma'')^{2^{l(b'-k_1)-1}}$ as another square root of A (because $b' > k_2 > k_1 \geq 1$, $b' - k_1 - 1 \geq 1$). If $a_2 \equiv \pm a_1$ then $\mathcal{B}$ fails to factor $n$; else it computes $h \leftarrow \gcd(a_1 - a_2, n)$ and outputs $(h, n/h)$ as factors of $n$.

Now we only need to prove that $a_2$ is a square root of A. Because $\sigma_{1,i}$ is a valid AR-$FssAgg^1$ signature, according to the verification process, we have the following equation:

$$[(\sigma')^{2^{l(T-k_2+1)}} \cdot y \cdot u^{c_{k_2}}]^{2^{l(k_2-k_1)}} \cdot y \cdot u_j^{c_{k_1}} = 1$$

That equals:

$$(\sigma'')^{2^{l(T-k_1+1)}} = y^{2^{l(k_2-k_1)+1}} \cdot u^{c_{k_2} \cdot 2^{l(k_2-k_1)} + c_{k_1}}$$

Because $u = s_{b'}^{2^{l(T+1-b')}}$ and $y = r_{b'}^{2^{l(T+1-b')}}$, the above equation can be written as:

$$\begin{aligned} (\sigma'')^{2^{l(T-k_1+1)}} &= (r_{b'} \cdot s_{b'}^{c_{k_2} \cdot 2^{l(k_2-k_1)} + c_{k_1}})^{2^{l(T+1-b')}} \\ &= A^{2^{l(T+1-b')}} \end{aligned} \quad (2)$$

Because $T - k_1 + 1 > T + 1 - b'$ and $A$ is a square, according to Lemma 1, we have

$$(\sigma'')^{2^{l(b'-k_1)}} = A$$

Therefore $a_2^2 = ((\sigma'')^{2^{l(b'-k_1)-1}})^2 = A$. That is, $a_2$ is a square root of $A$.

Algorithm $\mathcal{B}$ makes as many as signature queries as $\mathcal{A}$ makes. Algorithm $\mathcal{B}$'s running time is that of $\mathcal{A}$, plus the overhead in handling $\mathcal{A}$'s signature queries.

If $\mathcal{A}$ succeeds with probability of $\epsilon$ in forging, $\mathcal{B}$ succeeds at least with probability roughly $\epsilon/T$. The argument is outlined as follows. First, the view of $\mathcal{A}$ that $\mathcal{B}$ produces is computationally indistinguishable from the view of $\mathcal{A}$ interacting with a real $FssAgg^1$ signing oracle. Conditioned on $\mathcal{B}$ choosing the value of $b'$ as the period for which $\mathcal{A}$ eventually output a forgery, we have the probability that $\mathcal{B}$ outputs two factors is the same probability that $\mathcal{A}$ succeeds in forging, i.e., probability $\epsilon$. Since choosing the "right" $b'$ happens with probability $1/T$ we get that $\epsilon/T$ is an approximate lower bound on the forging probability of $\mathcal{B}$.

□