Attacks on Secure Logging Schemes

Gunnar Hartung[⋆]

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

Abstract. We present four attacks on three cryptographic schemes intended for securing log files against illicit retroactive modification. Our first two attacks regard the LogFAS scheme by Yavuz et al. (Financial Cryptography 2012), whereas our third and fourth attacks break the BM- and AR-FssAgg schemes by Ma (AsiaCCS 2008).
All schemes have an accompanying security proof, seemingly contradicting the existence of attacks. We point out flaws in these proofs, resolving the contradiction.

Keywords: Log Files · LogFAS · FssAgg · Digital Signatures · Forward Security · Attack · Cryptanalysis

1 Introduction

Log files record user-actions and events in computer systems, providing valuable information for intrusion detection, after-the-fact digital forensics, as well as system maintenance. For all of these objectives, having reliable information is imperative. Therefore, a number of historical and contemporary works on system security (e.g. [10, p. 10], [16, Sects. 18.3, 18.3.1], [7, Sect. 8.6]) recommend or require that log files be protected from unauthorized or retroactive modification.

It is generally desirable to use dedicated hardware (e.g. *write-once read many times drives*, so-called *WORM* drives) for this task, since such hardware can actually *prevent* the modification of log data. However, such special-purpose hardware is not always available. Therefore, cryptographers have devised schemes to provide integrity checks for log files that can purely be implemented in software. Such mechanisms can not prevent the manipulation of log data in the first place, but must be able to discern correct from manipulated information. The cryptographic schemes must retain their functionality *even if* an attacker has broken into the system and obtained the secret key. In order to achieve this, cryptographers have resorted to schemes (e.g. [5], [17], [8], [13], [11], [21], [22], [14]) that do not use a single secret key to authenticate information, but use a

[⋆] The research project leading to this report was funded by the German Federal Ministry of Education and Research under grant no. 01|S15035A. The author bears the sole responsibility for the content of this report. This paper will appear in the proceedings of the Financial Cryptography and Data Security conference 2017.

sequence of secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_T$ instead.[1] Each key sk_i is used for some time period (called the *i-th epoch*), until it is eventually replaced by its successor. In the following, we will focus on digital signature schemes, though MAC schemes using such a key-chain are used as well.

Informally speaking, a cryptographic signature scheme is called *forward-secure* if no attacker, who is given signatures on messages of his choice as well as a secret key sk_i from the sequence, can forge a signature relating to an epoch *before* the key-compromise. If a forward-secure signature scheme is used to sign log entries, an attacker breaking into the system during some epoch i will not be able to modify log entries from previous epochs $j < i$ without this change being detectable. (The attacker may, however, be able to arbitrarily modify log entries from later epochs. But since the attacker is in control of the *input* to the logging system once he has corrupted the signer, the attacker could control the log file's content even if the cryptographic scheme somehow prevented him from computing a signature.)

Since a log file will accumulate log entries over a possibly long period of time, the number of signatures being stored to verify the log messages will grow accordingly. For efficiency reasons, it is therefore desirable to be able to "compress" the signatures. *Aggregate signature schemes* [6] allow the signer to merge signatures on different messages (possibly even originating from different signers) into just one signature, which may be as small as a signature for a single message. Using aggregate signatures for secure logging does not only improve the logging system's efficiency, but also helps preventing so-called truncation attacks [13].

A special, but restricted case of aggregation is *sequential aggregation*. Sequential aggregation demands that aggregation/compression must be done at the time of creating a new signature. Ad-hoc aggregation of signatures that have been created independently needs not be supported. Ma and Tsudik [12] introduced the abbreviation "FssAgg" for **f**orward-**s**ecure **s**equential **agg**regate signatures.

The LogFAS scheme [22] as well as the BM-FssAgg and AR-FssAgg [11] schemes are modern constructions for securing log files. Both try to attain forward-security and aggregation, and were published on notable and peer-reviewed conferences.

Our Contribution. We describe two attacks on LogFAS [22,23], which allow for virtually arbitrary log file forgeries and for the confusion of legitimate signers, respectively, in Section 2. Our attacks on LogFAS have been acknowledged in private communication by one of authors of [22].

Furthermore, we present two attacks against the BM-FssAgg and AR-FssAgg schemes [11], which even allow for recovery of the signing key sk_i for specific epochs i. Our findings are given in Section 3. We implemented these attacks to verify our findings and to determine the required effort. We found that our first

[1] For efficiency reasons, schemes where each secret key can be computed from the previous one, and where there is only single, compact key for verification are desirable. However these properties are not strictly required.

attack on the BM-FssAgg scheme takes (depending on the parameters) between two and fifty minutes of computation, even with an implementation that misses a number of rather obvious optimizations. Our attack on AR-FssAgg required less than 0.05 seconds in all of our experiments. We present our experimental results in Section 3.7.

While LogFAS is a rather recent scheme, the BM- and AR-FssAgg schemes have been proposed several years ago. Nonetheless, the attacks we present have not been brought to public attention.

All three schemes have an accompanying security proof, which should rule out any meaningful attack on the schemes. We analyzed these proofs and identified a flaw in each of them, resolving the contradiction between our findings and the claimed security properties of the schemes. Note that our second attack on LogFAS is outside the security model considered in [22]; it therefore does not contradict the claimed security.

2 LogFAS

LogFAS [22] is a recently proposed forward-secure and aggregate audit log scheme. It offers high computational efficiency and compact public key sizes at the expense of large secret keys.

Before we describe our attacks, we will briefly introduce LogFAS. The reader is referred to [22,23] for a more detailed presentation.

2.1 Description of LogFAS

Let G be a subgroup of prime order q of $\mathbb{Z}_p^*$, where p is a prime number such that q divides $p-1$. Let α be a generator of G and T be the total number of supported epochs. LogFAS assumes a Key Generation Center (KGC) that generates keys for individual signers. Each signer i has an identity ID_i. Signatures of the LogFAS scheme consist of several values, some of which can be aggregated. For the remainder of this section, we employ the convention that variables with two indices are aggregated values of several epochs. For instance, $s_{0,l}$ is the aggregation of the values $s_0, \ldots, s_l$.

LogFAS uses three fundamental building blocks: an ordinary signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, the Schnorr signature scheme [18,19] (briefly recapped in Appendix A), and an incremental hash function IH based on a collision-resistant hash function H, which is modelled as a random oracle [4].

The key of IH consists of T factors $z_0, \ldots, z_{T-1}$. The hash value of a sequence of $l \in \{0, \ldots, T-1\}$ messages $(m_0, \ldots, m_l)$ is then given by

$$IH(m_0, \ldots, m_l) := \sum_{i=0}^{l} H(m_i) z_i \pmod{q} \ .$$

The security of this hash function can be shown under subset-sum-style assumptions, see the references in [22,23] for details.

An individual signer's secret key is derived from a central long-term secret $b \in \mathbb{Z}_q^*$ held by the KGC (which can be compared to a secret key of the Schnorr scheme) and several values chosen uniformly at random. Each signer's secret key includes a set of coefficients $z_0, \dots, z_{T-1}$ (derived from b) that form the key of *IH*. The exact relations between the values in the secret key, the public key and the signature are a little complicated, but our attack can be understood without fully comprehending how these values relate to each other.

The algorithms used by LogFAS are given below.

Key Generation.

The KGC chooses a random value $b \in \mathbb{Z}_q^*$ and generates a key pair $(\widehat{\mathsf{sk}}, \widehat{\mathsf{pk}})$ using Σ. The long term private and public keys are $(b, \widehat{\mathsf{sk}})$ and $(B := \alpha^{b^{-1} \pmod q}, \widehat{\mathsf{pk}})$, respectively. These values are shared for all signers.

Next, for each identity ID_i, the KGC generates temporary keys for each epoch $j \in \{0, \dots, T-1\}$ based on random values $r_j, a_j, d_j, x_j \leftarrow \mathbb{Z}_q^*$. These values are used to create interdependent variables as follows:

$$\begin{aligned} y_j &:= a_j - d_j && \pmod q, \\ z_j &:= (a_j - x_j)b && \pmod q, \\ M_j &:= \alpha^{x_j - d_j} && \pmod p, \quad \text{and} \\ R_j &:= \alpha^{r_j} && \pmod p. \end{aligned}$$

Finally, the KGC generates "tokens" $\beta_j \leftarrow \mathsf{Sign}(\widehat{\mathsf{sk}}, H(\mathrm{ID}_i \,\|\, j))$ for each signer ID_i and each epoch number j. These serve as witnesses that signer ID_i has created at least j signatures. Let $\mathsf{sk}'_i := (r_i, y_i, z_i, M_i, R_i, \beta_i)$ for each $i \in \{0, \dots, T-1\}$. The initial secret key of ID_i is $\mathsf{sk}_0 = \langle \mathsf{sk}'_0, \dots, \mathsf{sk}'_{T-1} \rangle$.

Key Update.

Updates the key sk_l ($l \in \{0, \dots, T-2\}$) to the next epoch sk_{l+1} by simply erasing r_l, y_l, M_l, and β_l from sk'_l.

Signature Generation.

A LogFAS signature $\sigma_{0,l-1}$ consists of aggregate-so-far values $s_{0,l-1} \in \mathbb{Z}_q$ and $M'_{0,l-1} \in \mathbb{Z}_p^*$, the most recent token β_{l-1}, as well as the random group elements R_j and the elements z_j of *IH*'s key for all $j \in \{0, \dots, l-1\}$.[2]

Given an aggregate signature $\sigma_{0,l-1}$ for $\langle m_0, \dots, m_{l-1} \rangle$, a new entry m_l and the temporary secret key $(r_l, y_l, z_l, M_l, R_l, \beta_l)$ for epoch l, first compute the hash value $e_l := H(m_l \,\|\, l \,\|\, z_l \,\|\, R_l)$. Then compute $s_l := r_l - e_l y_l \pmod q$ and aggregate this value into $s_{0,l} := s_{0,l-1} + s_l \pmod q$. Next, set $M'_l := M_l^{e_l} \pmod p$ and aggregate this into $M'_{0,l} := M'_{0,l-1} M'_l \pmod p$. The new aggregate signature is

$$\sigma_{0,l} := (s_{0,l}, M'_{0,l}, \beta_l, \langle (R_j, z_j) \rangle_{j=0}^{l}) \ .$$

Verification.

To verify an aggregate signature $(s_{0,l}, M'_{0,l}, \beta_l, \langle (R_j, z_j) \rangle_{j=0}^{l}) = \sigma_{0,l}$ over $l+1$

[2] The original scheme in [22] includes the value e_j in the signature. We have omitted this, as e_j can be recomputed by the verifier.

log entries $\langle m_0, \ldots, m_l \rangle$, one first checks the validity of the token β_l. If $\mathsf{Verify}(\widehat{\mathsf{pk}}, H(\mathrm{ID}_i \,\|\, l), \beta_l) = 0$, then output 0 and exit. Otherwise, compute $z_{0,l} := IH(m_0 \,\|\, 0 \,\|\, z_0 \,\|\, R_0,\ \ldots,\ m_l \,\|\, l \,\|\, z_l \,\|\, R_l)$, check if

$$\prod_{j=0}^{l} R_j \overset{?}{\equiv} M'_{0,l} \cdot B^{z_{0,l}} \cdot \alpha^{s_{0,l}} \pmod{p} \tag{1}$$

and accept if the equation holds (output 1 and exit). Otherwise, reject the signature (output 0 and exit).

2.2 The Attacks

We report two simple and efficient attacks on LogFAS. The first one allows for virtually arbitrary modification of log entries, but can not change the log file size. It requires only minimal computation and a single signature. This attack contradicts the claimed security of LogFAS. We analyzed the proof of security in [23] and found a flaw, resolving this contradiction.

Our second attack allows an attacker to masquerade a signature as originating from another (valid) signer. This attack is outside the formal security model considered in [22], and therefore does not contradict the claimed security. It nonetheless presents a serious threat, as it undermines the signature's authenticity.

Signature Forgery. Our first attack can be used to sign any sequence of log messages $\langle m_0^*, \ldots, m_l^* \rangle$ ($l \in \{0, \ldots, T-1\}$), provided the attacker has a valid signature for some other sequence of log messages $\langle m_0, \ldots, m_l \rangle$ of the same length, and knows the public key pk.

On a high level, our attack exploits the fact that the right hand side of Eq. (1) can be fully determined $M'_{0,l}$. Since $M'_{0,l}$ is part of the signature, an attacker can simply set $M'_{0,l}$ to a value such that the equation holds. Computing the respective value essentially only requires modular multiplication, exponentiation and inversion, which can be implemented quite efficiently.

Concretely, let $\sigma_{0,l} = (s_{0,l}, M'_{0,l}, \beta_l, \langle (R_j, z_j) \rangle_{j=0}^{l})$ be the signature known to the attacker. At first, the adversary computes $R_{0,l} = \prod_{j=0}^{l} R_j \pmod{p}$, and $z_{0,l} = IH(m_0^* \,\|\, 0 \,\|\, z_0 \,\|\, R_0,\ \ldots,\ m_l^* \,\|\, l \,\|\, z_l \,\|\, R_l)$. (S)he then sets $M^*_{0,l} := R_{0,l} \cdot B^{-z_{0,l}} \cdot \alpha^{-s_{0,l}} \pmod{p}$. The forged signature is $\sigma^*_{0,l} = (s_{0,l}, M^*_{0,l}, \beta_l, \langle (R_j, z_j) \rangle_{j=0}^{l})$.

It is easy to see that this signature will be accepted by the verification algorithm. Since β_l is taken from the original signature, it is a valid signature for $H(\mathrm{ID}_i \,\|\, l)$ and so $\mathsf{Verify}(\widehat{\mathsf{pk}}, H(\mathrm{ID}_i \,\|\, l), \beta_l)$ will return 1, i.e. the first check of the verification algorithm will succeed. Now, by our setup, we have

$$M^*_{0,l} \cdot B^{z_{0,l}} \cdot \alpha^{s_{0,l}} \equiv (R_{0,l} \cdot B^{-z_{0,l}} \cdot \alpha^{-s_{0,l}}) \cdot B^{z_{0,l}} \cdot \alpha^{s_{0,l}} \equiv R_{0,l} \equiv \prod_{j=0}^{l} R_j \pmod{p} \ .$$

Therefore, the verification algorithm will accept the signature, and the attack is successful. Note that the attack only replaces a single component of the signature, namely $M'_{0,l}$. All other parts of the signature are copied without modification. This simple attack is possible due to the structure of Eq. (1) , where the right hand side can be fully determined by $M'_{0,l}$ and this requires only modular multiplication, exponentiation and inversion.

Sender Confusion. If an attacker has two aggregate signatures $\sigma_{0,l}$, $\sigma'_{0,l}$ for two sequences of log messages of the same length $l+1$, created by different signers ID_i, $\mathrm{ID}_{i'}$ the attacker can just exchange the β_l tokens. The receiver will accept $\sigma_{0,l}$ as a signature from $\mathrm{ID}_{i'}$, when the messages were really signed by signer i, and vice versa. This attack is due to the fact that the identity ID_i of the signer is only bound to β_j but not to the other signature components $s_{0,l}, M'_{0,l}, R_j, z_j$.

2.3 Attack Consequences

In this section we present a scenario that shows how our attacks might be used in a real-world attack. Consider a corporate network, where there are multiple servers $S_1, \ldots S_n$ ($n \in \mathbb{N}$) offering different services. Each server S_i collects information in its log files, and regularly transfers all new log entries together with a signature to some central logging server L. The logging server L checks the signatures, stores the log data, and might examine it automatically for signs of a security breach using an intrusion detection system (IDS). If a server S_i does not transmit any new log entries to L within a certain amount of time, L raises an alarm (as there might be an attacker suppressing the delivery of log messages to L). Assume that LogFAS is used for signing log entries.

An attacker who has broken into a server S_i in the corporate network without raising an alarm might retroactively change the log entries not yet transmitted to L to cover his traces, and then create a new (valid) signature for the modified log file using our first attack. He continues to transmit log entries to L regularly, in order not to raise an alarm, albeit he suppresses log entries that might raise suspicion.

Now, assume that the attacker can bring himself into a man-in-the-middle position between some other server S_j and L. (This might be achieved using techniques such as ARP spoofing.) He may now filter and change log entries sent from S_j to L on-the-fly, while our first attack allows him to create valid signatures. Thus, the attacker may attack S_j without risking detection by the IDS at L.

To illustrate our second attack, suppose that the logging system was fixed to prevent the signature forgery. However, bringing himself into a man-in-the-middle position again, the attacker might still exchange the identities of some servers S_j, S_k included in the signature using our sender confusion attack. He may then try to compromise S_j, while the IDS raises an alarm regarding an attack on S_k. The attacker can thus misdirect the network administrators' efforts to defend their network, giving him an advantage, or at least gaining time until the administrators notice the deception.

2.4 The Proof of Security

In this section we point out the mistake in LogFAS' proof of security that allowed for the false conclusion of LogFAS being secure. The reader is advised to consider [23] while reading this section, or to skip this section entirely.

The security proof for LogFAS follows a simple and mostly standard scheme. One assumes an attacker $\mathcal{A}$ that breaks LogFAS, and constructs an attacker $\mathcal{F}$ against the Schnorr signature scheme, using $\mathcal{A}$ as a subroutine. $\mathcal{F}$ first guesses an index w of a message block that $\mathcal{A}$ will modify. $\mathcal{F}$'s challenge public key (for the Schnorr scheme) is then embedded into the temporary key pair for that message, the remaining key pairs are set up honestly.

When the attacker outputs a forgery, the proof considers three cases. The first case deals with attackers that actually create a new message together with a valid signature (as does our attack). The second case deals with truncation attacks and the third case models a hash collision.

The error is located in the first case, where the authors conclude that a forgery for an entirely new message must imply a forgery of a Schnorr-type signature, i.e. that the values R_w, s_w (when properly extracted from the LogFAS signature) must be a valid signature for the message m_w. We can see that this conclusion is false, since our attack does not modify the values R_w, s_w at all, but only replaces the original message with an arbitrary one. Thus, the verification algorithm of the Schnorr scheme will reject the signature with very high probability, while the authors conclude that the signature will be accepted.

3 The FssAgg Schemes

This section presents the BM-FssAgg scheme, the AR-FssAgg scheme and our attacks on these constructions. Both schemes were presented in [11], and are intended to provide a single signature per epoch. Thus, the respective key must be updated every time a message has been signed.

3.1 Description of the BM-FssAgg Scheme

The BM-FssAgg signature scheme [11] is based on a forward-secure signature scheme by Bellare and Miner [3]. Both schemes utilize repeated squaring modulo a Blum integer N. (An integer N is called a *Blum integer* if it is a product of two primes p, q such that $p \equiv q \equiv 3 \pmod 4$.) Again, we first describe the BM-FssAgg scheme before we turn to our attack.

Let T be the number of supported epochs and H a hash function that maps arbitrary bit strings to bit strings of some fixed length $l \in \mathbb{N}$.

Intuitively, the scheme is built on $l+1$ sequences of units modulo N, where in each sequence, each number is obtained by squaring the predecessor. Once the starting points r_0 and $s_{i,0}$ (for $i \in \{1,\dots,l\}$) have been selected during key generation, the scheme successively computes

$$\begin{aligned} r_{j+1} &:= r_j^2 \pmod N \quad \text{for } j \in \{0,\dots,T\} \\ s_{i,j+1} &:= s_{i,j}^2 \pmod N \quad \text{for } j \in \{0,\dots,T\} \text{ and } i \in \{1,\dots,l\}\,. \end{aligned} \tag{2}$$

When r_0 and the $s_{i,0}$ are clear from the context, we may thus naturally refer to r_j and $s_{i,j}$ for $j \in \{1, \ldots, T+1\}$ throughout this section. Observe that these sequences form one-way chains: Given any element $s_{i,j}$ of a chain, it is easy to compute the subsequent elements $s_{i,j'}$ with $j' > j$, but it is unknown how to efficiently compute the previous ones without knowing the factorization of N. (Obviously, the same holds for the chain of the r_j-s.)

We now describe the BM-FssAgg scheme in more detail.

Key Generation.
Pick two random, sufficiently large primes p, q, each congruent to 3 modulo 4, and compute $N = pq$. Next, pick $l+1$ random integers $r_0, s_{1,0}, \ldots s_{l,0} \leftarrow \mathbb{Z}_N^*$. Compute $y := 1/r_{T+1} \pmod N$, and $u_i := 1/s_{i,T+1} \pmod N$ for all $i \in \{1, \ldots, l\}$. The public key is then defined as $\mathsf{pk} := (N, T, u_1, \ldots u_l, y)$, whereas the initial secret key is $\mathsf{sk}_1 := (N, j = 1, T, s_{1,1}, \ldots, s_{l,1}, r_1)$.

Key Update.
In order to update the secret key, simply replace all $r_j, s_{i,j}$ by the respective $r_{j+1}, s_{i,j+1}$ (i.e., square all these values), and increment the epoch counter j.

Signing.
In order to sign a message m_j, first compute the hash value $c := H(j, y, m)$. Let $c_1, \ldots, c_l \in \{0, 1\}$ be the bits of c. The signature for m is $\sigma_j := r_j \prod_{i=1}^{l} s_{i,j}^{c_i}$, i.e., the signature is the product of r_j and all $s_{i,j}$ where $c_i = 1$. An aggregate signature for multiple messages is computed by multiplying the individual signatures. Thus, a signature can be added to an aggregate signature $\sigma_{1,j-1}$ by computing the new aggregate as $\sigma_{1,j} = \sigma_{1,j-1} \cdot \sigma_j \pmod N$.

Verification.
Given an aggregate signature $\sigma_{1,t}$ for messages $m_1, \ldots, m_t$ signed in epochs 1 through t, the verification algorithm will effectively "strip off" the individual signatures one-by-one, starting with the *last* signature.

More precisely, to verify $\sigma_{1,t}$, act as follows: Recompute the hash value $c_t = c_{1,t} \ldots c_{l,t} := H(t, y, m_t)$ of the last message. (Recall that the signature for m_t is $r_t \prod_{i=1}^{l} s_{i,t}^{c_{i,t}}$.) Square $\sigma_{1,t}$ exactly $T + 1 - t$ times, effectively adding $T + 1 - t$ to the j-indices of all r_j, $s_{i,j}$ contained in $\sigma_{1,t}$. (In particular, this effectively changes the signature for m_t to $r_{T+1} \prod_{i=1}^{l} s_{i,T+1}^{c_{i,T+1}}$.) Multiply the result with $y \prod_{i=1}^{l} u_i^{c_{i,t}}$, cancelling out the last signature because y and the u_i are the modular inverses of r_{T+1} and the $s_{i,T+1}$.

For the last-but-one message, square the result another time (projecting the last-but-one signature into the epoch $T + 1$), recompute the hash value $c_{1,t-1} \ldots c_{l,t-1}$, and cancel out the last-but-one signature by multiplication with $y \prod_{i=1}^{l} u_i^{c_{i,t-1}}$.

The scheme continues analogously for the remaining messages $m_{t-2}, \ldots, m_1$. If the procedure terminates at a value of 1, the aggregate signature is accepted as valid, otherwise it is rejected as invalid.

3.2 Attack on the BM-FssAgg Scheme

We show a conceptually simple way to recover the secret key sk_t ($t \geq l+1$) from t successive aggregate signatures and the public key pk. (Our attack may work with $t = l+1$ signatures, but has a higher success probability if $t > l+1$. In our experiments, $t = l+11$ signatures have been sufficient for all cases.)

Our attack makes use of the fact that the r_j values, which are supposed to randomize the signatures, are not chosen independently at random, but are strongly interdependent.[3] This allows us to set up a set of equations with a limited number of variables (namely, r_t and the $s_{i,t}$), and then solve the equations for these variables, which together make up the secret key sk_t.

We will now describe our attack in more details. Fix arbitrary messages $m_1, \ldots, m_t$ and the respective aggregate signatures $\sigma_{1,j}$, each valid for messages $m_1, \ldots, m_j$. Let $c_{i,j}$ denote the i-th bit of the hash value of message m_j, as computed by the signing algorithm.

First, recover the individual signatures $\sigma_j := \sigma_{1,j}/\sigma_{1,j-1} \pmod{N}$ for all $j \in \{1, \ldots, t\}$, letting $\sigma_{1,0} = 1$. Observe that

$$\begin{array}{ccccccc} \sigma_1 &=& r_1 & s_{1,1}^{c_{1,1}} & \ldots & s_{l,1}^{c_{l,1}} \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ \sigma_t &=& r_t & s_{1,t}^{c_{1,t}} & \ldots & s_{l,t}^{c_{l,t}} . \end{array}$$

For ease of presentation, we let $s_{0,j} = r_j$ and $c_{0,j} = 1$ for all j. We define $\tau_j := \sigma_j^{(2^{t-j})}$, i.e. we square each signature σ_j for $t-j$ times, effectively adding $t-j$ to the j-index of the $r_j, s_{i.j}$ because of Eq. (2). We thus obtain

$$\begin{array}{ccccccccc} \tau_1 &=& \sigma_1^{(2^{t-1})} &=& s_{0,t}^{c_{0,1}} & s_{1,t}^{c_{1,1}} & \ldots & s_{l,t}^{c_{l,1}} \\ \tau_2 &=& \sigma_2^{(2^{t-2})} &=& s_{0,t}^{c_{0,2}} & s_{1,t}^{c_{1,2}} & \ldots & s_{l,t}^{c_{l,2}} \\ \vdots & & \vdots & & \vdots & \vdots & \ddots & \vdots \\ \tau_{t-1} &=& \sigma_{t-1}^{(2^1)} &=& s_{0,t}^{c_{0,t-1}} & s_{1,t}^{c_{1,t-1}} & \ldots & s_{l,t}^{c_{l,t-1}} \\ \tau_t &=& \sigma_t^{(2^0)} &=& s_{0,t}^{c_{0,t}} & s_{1,t}^{c_{1,t}} & \ldots & s_{l,t}^{c_{l,t}} , \end{array} \tag{3}$$

where all $c_{i,j} \in \{0,1\}$. We thus have $t \geq l+1$ equations in the $l+1$ unknown variables $s_{i,t}$. We now want to solve these equations for the $s_{i,t}$, by doing linear algebra "in the exponent". We can later realize addition and subtraction of row vectors $(c_{0,j}, \ldots, c_{l,j})$ by multiplication and division of the τ_j, respectively. Likewise, multiplication of a row vector by a scalar $z \in \mathbb{Z}$ can be realized by raising the respective τ_j to its z-th power.

More concretely, we consider the $c_{i,j}$ as a matrix C over the integers, and try to express each standard basis vector e_i as an integer linear combination of the row vectors $c_j = (c_{0,j}, \ldots, c_{l,j})$.

[3] For this reason, our attack does not carry over to the underlying forward-secure signature scheme by Bellare and Miner [3]. There, the values r_j are chosen uniformly and independently at random, which prevents our attack.

Note that the Gaussian elminiation method is *not* suited for this setting, since it will compute a linear combination of the row-vectors if one exists, but the output may not be an *integer* linear combination. Moreover, a set of $l+1$ row vectors $(c_{0,j} \dots c_{l,j})$ may not form a basis of $\mathbb{Z}^{l+1}$ *even if* they are linearly independent, since the integers are not a field, and thus $\mathbb{Z}^{l+1}$ is not a vector space but only a $\mathbb{Z}$-module. (We will nonetheless continue to refer to elements of $\mathbb{Z}^{l+1}$ as "vectors" for simplicity.) We therefore need to employ different algorithms.

Specifically, we compute the *Hermite Normal Form* (HNF) of C. The exact definitions and conventions used for the HNF differ in the literature. The following definition is a special case of Definition 2.8 given by [2, p. 301], applying the preceding Example 2.7 (1) on the same page.

Definition 1. *Let $A \in \mathbb{Z}^{m\times n}$ be an integer matrix. Denote the i-th row of A by a_i, and the j-th entry of the i-th row by $a_{i,j}$ (for $i \in \{1,\dots,m\}$ and $j \in \{1,\dots,n\}$). A is in* Hermite Normal Form *iff there is a non-negative integer r with $0 \le r \le m$ such that*

1. *$a_i \neq 0$ for all $1 \le i \le r$ and $a_i = 0$ for all $r+1 \le i \le m$, and*
2. *there is a sequence of column indices $1 \le n_1 < \dots < n_r \le m$ such that for all $i \in \{1,\dots,r\}$ the following three conditions hold:*

$$\begin{aligned} a_{i,n_i} &> 0 \\ a_{i,j} &= 0 && \text{for } j < n_i, \text{ and} \\ 0 \le a_{j,n_i} &< a_{i,n_i} && \text{for } 1 \le j < i. \end{aligned}$$

Intuitively, a matrix is in HNF if only the first r rows are occupied (and the remaining $m-r$ rows are zero), each non-zero row has a positive "pivot" element a_{i,n_i} (which is the first non-zero element in this row), the pivot element of each row is further to the right than the pivot of the preceding row, and all elements *above* a pivot element are between 0 (inclusive) and the pivot (exclusive).

Each integer matrix A can be transformed into a matrix H in HNF by a set of invertible row operations, represented by a unimodular matrix R (i.e. $RA = H$) [2, Theorem 2.9, p. 302], and the HNF H of a given integer matrix A is unique [2, Theorem 2.13, p. 304]. Furthermore, the HNF is known to be computable in polynomial time, see e.g. [9,15].

Assume for now that the rows of C span $\mathbb{Z}^{l+1}$. (We will show that this is a realistic assumption given enough signatures in Section 3.7). If this is the case, then the HNF of C is

$$H = \begin{pmatrix} \mathbf{1}_{l+1} \\ \mathbf{0}_{t-(l+1),l+1} \end{pmatrix} \tag{4}$$

where $\mathbf{1}_{l+1}$ is the $(l+1)\times(l+1)$ identity matrix and $\mathbf{0}_{t-(l+1),l+1}$ is the all-zero matrix with $t-(l+1)$ rows and $l+1$ columns. In the following, let $e_i = (e_{i,1},\dots,e_{i,l+1}) \in \mathbb{Z}^{l+1}$ be the i-th unit vector. (Thus $e_{i,j} = 1$ if $i = j$, and $e_{i,j} = 0$ otherwise.)

Continuing our attack, we compute the matrix $R = (r_{i,j}) \in \mathbb{Z}^{t\times t}$ that transforms C into its Hermite Normal Form H (i.e., $RC = H$). We then fix

$i \in \{0, \dots, l\}$ and compute

$$\begin{aligned}\prod_{j=1}^{t} (\tau_j)^{r_{i,j}} &= (s_{0,t}^{c_{0,1}} \dots s_{l,t}^{c_{l,1}})^{r_{i,1}} \cdot \dots \cdot (s_{0,t}^{c_{0,t}} \dots s_{l,t}^{c_{l,t}})^{r_{i,t}} \\ &= (s_{0,t}^{r_{i,1}c_{0,1}+\dots+r_{i,t}c_{0,t}}) \cdot \dots \cdot (s_{l,t}^{r_{i,1}c_{l,1}+\dots+r_{i,t}c_{l,t}}) \\ &= s_{0,t}^{e_{i,0}} \cdot \dots \cdot s_{l,t}^{e_{i,l}} \\ &= s_{i,t}\end{aligned}$$

where the first equality follows from substituting the signatures according to Eq. (3) and writing out the product, and the second equality can be obtained by sorting the product by the base terms. To see the third and fourth equality, note that the exponents for the $s_{i,t}$ match the i-th row of the matrix $RC = H$, and that the first $l+1$ rows of H are the unit vectors (see Eq. (4)).

Overall, this gives away $s_{i,t}$. Repeating this step for all $i \in \{0, \dots, l\}$ allows us to reconstruct all $s_{i,t}$, thus leaking the entire secret key sk_t of the t-th epoch. This concludes the description of our attack against BM-FssAgg.

3.3 Description of the AR-FssAgg Scheme

The AR-FssAgg scheme by Ma [11] is based on a forward-secure digital signature scheme by Abdalla and Reyzin [1], which itself is based on the forward-secure signature scheme by Bellare and Miner [3], but is considerably more efficient.

In the following, we will briefly describe the differences between the AR-FssAgg scheme and the BM-FssAgg scheme. The reader is referred to [11] for a complete description of the AR-FssAgg construction.

The main difference between the AR-FssAgg scheme and the BM-FssAgg scheme is that the former interprets the hash function's output c as an integer in $[0, 2^l - 1]$. Consequently, the $l+1$ chains of squares $r_j, s_{i,j}$ are replaced by just two chains r_j, s_j of higher powers, namely:

$$\begin{aligned} r_{j+1} &:= r_j^{(2^l)} \pmod N \qquad \text{for } j \in \{0, \dots, T\} \\ s_{j+1} &:= s_j^{(2^l)} \pmod N \qquad \text{for } j \in \{0, \dots, T\}\,. \end{aligned}$$

As for the BM-FssAgg scheme, the starting points r_0 and s_0 are chosen randomly, and N is a Blum integer. The key update procedure is adapted canonically: r_j and s_j are raised to their 2^l-th power instead of being squared. Thus, they are replaced by $r_j^{2^l}$ and $s_j^{2^l}$, respectively. In the signing procedure, the hash value c is computed as before, but the signature for the single message is now $\sigma_j := r_j \cdot s_j^c \pmod N$. The aggregate signature is again $\sigma_{1,j} := \sigma_{1,j-1} \cdot \sigma_j \pmod N$, as before.

3.4 Attack on the AR-FssAgg Scheme

As with the BM-FssAgg scheme, our attack on the AR-FssAgg scheme allows us to reconstruct the secret key of a particular epoch t ($t \geq 3$), requiring only the public key and a few consecutive aggregate signatures $\sigma_{1,1}, \ldots, \sigma_{1,t}$. Our attack again exploits the fact that the supposedly random values r_j are not actually chosen independently at random, but depend on each other.[4]

As before, we first recover the individual signatures as $\sigma_j := \sigma_{1,j}/\sigma_{1,j-1} \pmod N$ for $j \in \{1, \ldots, t\}$, and "project" them into the epoch t, by computing: $\tau_j = \sigma_j^{2^{l(t-j)}}$. We again obtain a system of equations:

$$\begin{array}{cccc} \tau_1 = \sigma_1^{2^{l(t-1)}} & = r_t \cdot s_t^{c_1} \\ \vdots \quad\quad \vdots & \vdots \quad \vdots \\ \tau_t = \quad \sigma_t^{(2^0)} & = r_t \cdot s_t^{c_t} \end{array} \tag{5}$$

We pick one of the τ_j arbitrarily, say τ_1, and use it to strip the r_t from the other τ_j-s, by computing $\phi_j := \tau_j/\tau_1 = s^{c_j}/s^{c_1} = s^{c_j - c_1}$ for all $j \in \{2, \ldots, t\}$.

For brevity, let $c'_j = c_j - c_1$ for all $j \in \{2, \ldots t\}$. We assume for now that the greatest common divisor of all c'_j is 1. (We will revisit this assumption in Section 3.7.)

Once we have the ϕ_j, we use the extended Euclidean algorithm to obtain coefficients $f_2, \ldots, f_t$ such that $c'_2 f_2 + c'_3 f_3 + \ldots + c'_t f_t = \gcd(c'_2, c'_3, \ldots c'_t) = 1$. We can then compute $\phi_2^{f_2} \cdot \phi_3^{f_3} \cdots \phi_t^{f_t} = s_t^{c'_2 f_2} \cdot s_t^{c'_3 f_3} \cdots s_t^{c'_t f_t} = s_t^1 = s_t$.

Once we know s_t, r_t can be recovered trivially from (e.g.) τ_1, by computing $r_t := \tau_1 / s_t^{c_1} \pmod N$. We have thus recovered the secret key for epoch t.

3.5 Attack Consequences

Reconsider the scenario from Section 2.3, but assume that log entries are signed with the BM-FssAgg or AR-FssAgg scheme instead of LogFAS.

Assume again that an attacker has managed to break into a server S_i without raising an alarm. He may then bring himself into a man-in-the-middle position between another server S_j and L again, and first passively observes several transmissions of log entries from S_j to L, storing the respective signatures.

If at least t signatures for individual messages can be recovered from the (aggregate) signatures sent to L, the attacker can launch one of the attacks described above to recover a recent secret key.[5] He may then attack the server S_j, filtering the log messages sent from S_j to L on-the-fly, and create valid signatures using the known secret key.

While it may seem unnatural that the aggregate signatures observed by the attacker are directly consecutive, it is actually a plausible scenario. For example, this might happen when the server S_j is mostly idle, e.g. at night.

[4] As with our attack on the BM-FssAgg scheme, our attack does not carry over to the underlying forward-secure signature scheme by Abdalla and Reyzin [1], since the values r_j are chosen independently at random in their signature scheme.

[5] Our attacks can be easily generalized to work with any $t+1$ consecutive aggregate signatures $\sigma_{1,k}, \ldots, \sigma_{1,k+t+1}$ or even with any t pairs of directly consecutive aggregate signatures $(\sigma_{1,k_1}, \sigma_{1,k_1+1}), \ldots, (\sigma_{1,k_t}, \sigma_{1,k_t+1})$.

3.6 The Proofs of Security

Security proofs for the BM-FssAgg and AR-FssAgg schemes are given in the appendix of [11]. Both proofs give a reduction to the hardness of factoring a Blum integer, assuming an efficient forger $\mathcal{A}$ on the respective scheme, and constructing an attacker $\mathcal{B}$ on the factorization of N. The proofs are incorrect for they assume that not only $\mathcal{A}$ may use a signing oracle, but $\mathcal{B}$ has access to a signing oracle, too.

3.7 Experimental Results

We implemented our attacks on the BM-FssAgg and AR-FssAgg schemes in order to verify them, and to empirically determine the number t of signatures required. (Recall that we assumed that the matrix C spanned $\mathbb{Z}^{l+1}$ for the attack on BM-FssAgg, and that $\gcd(c'_2, \ldots, c'_t) = 1$ for the attack on AR-FssAgg, respectively.) We measured the run times of our attacks, and found that they are entirely practical.

Since the attacks require a number of signatures, we also implemented the key generation, key updating and signing procedures of the two schemes.[6] The implementations are written for the computer algebra system Sage [20].

Our attack implementations miss a number of quite obvious optimizations: we did not parellelize independent tasks, and some computations are repeated during the attacks. Our measurements should therefore not be regarded as a precise estimate of the resources required for the respective attacks, but as an upper bound.

Experiment Setup. All experiments used a modulus size of 2048 bit and were conducted on a desktop office PC, equipped with a four-core AMD A10-7850K Radeon R7 processor with a per-core adaptively controlled clock frequency of up to 3.7 GHz, different L1-caches with a total capacity of 256 KiB, two 2 MiB L2-Caches, each shared between two cores, and 14.6 GiB of RAM. The PC was running version 16.04 of the Ubuntu Desktop GNU/Linux operating system, Sage in version 6.7, and Python 2.7.

For our attack on the BM-FssAgg scheme, we used the SHA-224, SHA-256, SHA-384 and SHA-512 hash functions to examine the influence of the hash length l on the runtime of our attack. The BM-FssAgg scheme was instantiated with 512 epochs for the SHA-224, SHA-256 and SHA-384 hash functions, and with 1024 epochs for the SHA-512 hash function. (Recall that the scheme signs exactly one message per epoch, and our attack on the BM-FssAgg scheme requires at least l signatures, where l is the hash length.)[7]

[6] Our implementation of the schemes is *only* intended to provide a background for our attacks. We did therefore not attempt to harden our implementation against different types of attacks at all.

[7] The number of supported epochs T may be unrealistically low. But since T does not influence the time required for executing our attacks, a small T is sufficient for our demonstration.

Our implementation of the attacks first collects the minimum required number of signatures ($l + 1$ for the BM-FssAgg scheme, where l is the output length of the hash function, and 3 for the AR-FssAgg scheme), and then checks if the respective requirement on the hash values is fulfilled. If this is the case, the attack is continued as described above. Otherwise, our implementation gradually requests additional signatures until the requirements are fulfilled.

For both of our schemes, we measured the time that was necessary to collect the total number of signatures. (This includes the time necessary to compute the signatures in the first place, and to update the keys respectively.) For our BM-FssAgg implementation, this time also includes the computation of the Hermite Normal Form of the given matrix, along with the transformation matrix. For the AR-FssAgg attack, the time includes the computation of the gcd of the c'_i, as well as the factors f_i. We refer to these times as the *signature collection times*. The remaining time required for the attacks is referred to as *reconstruction time*. A *measurement* corresponds to one execution of an attack.

Our experiments quickly showed that the reconstruction times for BM-FssAgg were quite long. Given the large amount of time required for the reconstruction and the small amount of variation in the reconstruction times, we restricted our examination of the reconstruction times of BM-FssAgg to 50 measurements per hash-function. For the reconstruction time of the attack on the AR-FssAgg scheme, the number of requested signatures (for both schemes), and the the signature collection times (for both schemes), we collected 250 measurements per scheme and hash-function.

Results Our results are summarized in Table 1. All times are given in seconds.

In our experiments regarding the attack on BM-FssAgg, the greatest difference $d = t-(l+1)$ between t (the number of actual required signatures) and $l+1$ (the minimum number of required signatures) was 10. (So, $t = l + 11$ signatures were always sufficient.) For AR-FssAgg, $t = 3 + 4$ have been sufficient for all of our 250 tries. The number of signatures actually required in our experiments is shown in the top third of Table 1. The theoretical minimum of signatures required to launch the attacks is given for comparison, denoted as "Theoretical Optimum".

We found that despite the lack of optimizations, our attack on BM-FssAgg took only minutes to recover the respective secret key (in the case of SHA-224) and at most 50 minutes (in the case of SHA-512). Our attack on the AR-FssAgg scheme took less than 0.05 seconds in all 250 measurements.

For BM-FssAgg, the reconstruction time turned out to be the major part of the attack time. In retrospect, this is understandable, since the computation of a single $s_{i,t}$ requires t modular exponentiations, so the reconstruction of all $s_{i,t}$ (including $r_t = s_{0,t}$) required $t \cdot (l + 1) \geq (l + 1)^2$ modular exponentiations.

Scheme	BM-FssAgg				AR-FssAgg
Hash Function	SHA-224	SHA-256	SHA-384	SHA-512	SHA-256
Signatures Required					
Theoretical Optimum	225	257	385	513	3
Observed Minimum	226	258	386	514	3
Average	227.15	259.05	387.27	514.97	3.67
Standard Deviation	1.42	1.33	1.73	1.38	0.97
Maximum	234	264	395	522	7
Signature Collection Times					
Minimum	11.74	17.13	65.46	180.02	9.0e-3
Average	22.18	28.79	118.98	292.33	11e-3
Standard Deviation	9.88	12.34	62.34	136.14	3.0e-3
Maximum	67.87	76.59	430.97	1006.61	22e-3
Reconstruction Times					
Minimum	104.06	154.14	580.41	1502.37	6.0e-3
Average	121.48	170.81	634.34	1753.68	9.2e-3
Standard Deviation	9.09	17.17	48.24	126.57	4.4e-3
Maximum	137.94	207.54	736.52	1935.59	24e-3

Table 1. Experimental Results. All times are given in seconds.

4 Summary

We have presented four attacks on LogFAS [22], the BM-FssAgg scheme, and the AR-FssAgg scheme [11]. The attacks on LogFAS have been acknowledged by one of LogFAS' authors, and we have demonstrated the practicality of our attacks on BM-FssAgg and AR-FssAgg experimentally. Our attacks allow for virtually arbitrary forgeries, or even reconstruction of the secret key. We conclude that neither of these schemes should be used in practice. If one of these should already be in use, we suggest immediate replacement.

Acknowledgements

I'd like to thank Alexander Koch for his detailed comments, as well as for questioning the security proof of the BM-FssAgg scheme, which was the starting point for my research presented in Section 3.

References

1. Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, vol-

ume 1976 of *Lecture Notes in Computer Science*, pages 116–129. Springer Berlin Heidelberg, 2000.

2. William A. Adkins and Steven H. Weintraub. *Algebra: An Approach via Module Theory*, volume 136 of *Graduate Texts in Mathematics*. Springer, New York, 1992.
3. Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer Berlin Heidelberg, 1999.
4. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. ACM.
5. Mihir Bellare and Bennet S. Yee. Forward integrity for secure audit logs. Technical report, University of California at San Diego, 1997.
6. Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 416–432. Springer Berlin Heidelberg, 2003.
7. Common Criteria for Information Technology Security Evaluation, version 3.1 r4, part 2, September 2012. `https://www.commoncriteriaportal.org/cc/`.
8. Jason E. Holt. Logcrypt: Forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian Workshops on Grid Computing and e-Research – Volume 54*, ACSW Frontiers '06, pages 203–211, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
9. Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, 1979.
10. Donald C. Latham, editor. *Department of Defense Trusted Computer System Evaluation Criteria*. US Department of Defense, December 1985. `http://csrc.nist.gov/publications/history/dod85.pdf`.
11. Di Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 341–352, New York, NY, USA, 2008. ACM.
12. Di Ma and Gene Tsudik. Forward-secure sequential aggregate authentication. Cryptology ePrint Archive, Report 2007/052, 2007. `http://eprint.iacr.org/`.
13. Di Ma and Gene Tsudik. A new approach to secure logging. In Vijay Atluri, editor, *Data and Applications Security XXII*, volume 5094 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin Heidelberg, 2008.
14. Giorgia Azzurra Marson and Bertram Poettering. *Practical Secure Logging: Seekable Sequential Key Generators*, pages 111–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
15. Daniele Micciancio and Bogdan Warinschi. A linear space algorithm for computing the hermite normal form. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, ISSAC '01, pages 231–236, New York, NY, USA, 2001. ACM.
16. An introduction to computer security: The NIST handbook, October 1995. NIST Special Publication 800-12.
17. Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *The Seventh USENIX Security Symposium Proceedings*, 1998.

18. Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer New York, 1990.
19. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
20. William Stein. Sagemath. `http://www.sagemath.org/`, last accessed 2016-10-25.
21. Attila A. Yavuz and Ning Peng. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 219–228, Dec 2009.
22. Attila A. Yavuz, Ning Peng, and Michael K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin Heidelberg, 2012.
23. Attila A. Yavuz and Michael K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. Technical Report TR-2011-21, North Carolina State University. Department of Computer Science, September 2011. `http://www.lib.ncsu.edu/resolver/1840.4/4284`.

A The Schnorr Signature Scheme

The Schnorr Signature Scheme [18,19] is based on the hardness of the discrete logarithm problem in some group G. It uses a prime-order subgroup G of $\mathbb{Z}_p^*$, where p is large a prime, G's order q is also a large prime, and q divides $p-1$. Let α be a generator of G. A secret key for Schnorr's scheme is $y \leftarrow \mathbb{Z}_q^*$, the corresponding public key is $Y := \alpha^y \pmod{p}$.

In order to sign a message m, choose $r \leftarrow \mathbb{Z}_q^*$, set $R := \alpha^r \pmod{p}$, compute the hash value $e := H(m \,\|\, R)$ and set $s := r - ey \pmod{q}$. The signature is the tuple (R, s). To verify such a signature, recompute the hash value $e := H(m \,\|\, R)$ (where R is taken from the signature and m is given as input to the verification algorithm). Then check if $R = Y^e \alpha^s \pmod{p}$ and return 1 if and only if this holds.

The Schnorr signature scheme can be shown to be secure based on the hardness of the discrete logarithm problem in G, if H is modelled as a random oracle [4].