Sequential Aggregate Signatures from Trapdoor Permutations

Anna Lysyanskaya
anna@cs.brown.edu

Silvio Micali

Leonid Reyzin
reyzin@cs.bu.edu

Hovav Shacham
hovav@cs.stanford.edu

Abstract

An aggregate signature scheme (recently proposed by Boneh, Gentry, Lynn and Shacham) is a method for combining n signatures from n different signers on n different messages into one signature of unit length. We propose *sequential aggregate signatures*, in which the set of signers is ordered. The aggregate signature is computed by having each signer, in turn, add his signature to it. We show how to realize this in such a way that the size of the aggregate signature is independent of n. This makes sequential aggregate signatures a natural primitive for certificate chains, whose length can be reduced by aggregating all signatures in a chain. We give a construction based on families of certified trapdoor permutations, and show how to instantiate our scheme based on RSA.

1 Introduction

Authentication constitutes one of the core problems in cryptography. Much modern research focuses on constructing authentication schemes that are: (1) as secure as possible, i.e., provably secure under the most general assumptions; and (2) as efficient as possible, i.e., communication- and computation-efficient. For cryptographic schemes to be adopted in practice, efficiency is crucial. Moreover, communication and storage efficiency — namely, the size of the authentication data, for example the size of a signature — plays an even greater role than computation: while computational power of modern computers has experienced rapid growth over the last several decades, the growth in bandwidth of communication networks seems to have more constraints.

As much as we would like to reduce the size of a stand-alone signature, its length is roughly equal to the security parameter. The problem becomes more interesting, however, once we have n different signers with public keys $PK_1, \dots, PK_n$, and each of them wants to sign her own message, $M_1, \dots, M_n$, respectively. Suppose that the public keys and the messages are known to the recipient of the signature ahead of time: they may be clear from the context. We want, in some way, to combine the authenticating information associated with this set of signers and messages, into one short signature, whose length is independent of n.

This problem actually arises in practice. For example, in a Public Key Infrastructure (PKI) of depth n, a certificate on a user's public key consists of a chain of certificates issued by a hierarchy of certification authorities (CAs): the CA at depth i is certifying the CA at depth $i+1$. Which CAs were responsible for certifying a given user is usually clear from the context, and the public keys of these CAs may be available to the recipient off-line. The user's certificate, however, needs to be included in all of his communications, and therefore it is highly desirable to make its length independent of the length of the certification chain.

Recently, Boneh et al. [5] introduced and realized *aggregate signatures*. An aggregate signature scheme is a signature scheme which, in addition to the usual setup, signing, and verification algorithms, admits an efficient algorithm for *aggregating* n signatures under n different public keys into one signature of unit length. Namely, suppose each of n users has a public-private key pair (PK_i, SK_i); each wishes to attest to a message M_i. Each user first signs her message M_i, obtaining a signature σ_i; the n signatures can then be combined by an unrelated party into an aggregate σ. An aggregate signature scheme also includes an extra verification algorithm that verifies such an aggregate signature. An aggregate signature provides non-repudiation simultaneously on message M_1 for User 1, message M_2 for User 2, and so forth. Crucially, such repudiation holds for each user *regardless* of whether other users are malicious. Boneh et al. construct an aggregate signature scheme in the random oracle model under the bilinear Diffie-Hellman assumption (see, for example, Boneh and Franklin [4] and references therein).

However, for applications such as certificate chains, the ability to combine preexisting individual signatures into an aggregate is unnecessary. Each user, when producing a signature, is aware of the signatures above his in the chain. Thus aggregation for certificate chains should be performed incrementally and sequentially, so that User i, given an aggregate on messages $M_1, \ldots, M_{i-1}$ under keys $PK_1, \ldots, PK_{i-1}$, outputs an aggregate on messages $M_1, \ldots, M_{i-1}, M_i$ under keys $PK_1, \ldots, PK_{i-1}, PK_i$. We call such a procedure *sequential aggregation*, and a signature scheme supporting it, a *sequential aggregate signature scheme.*

In this paper, we begin by giving a formal definition of sequential aggregate signatures. We then show how to realize such signatures from a family of certified[1] trapdoor permutations (TDPs) over the same domain, as long as the domain is a group under some operation. We prove security (with exact security analysis) of our construction in the random oracle model; we give tighter security guarantees for the special cases of *homomorphic* and *claw-free* TDPs. As compared to the scheme of [5], our scheme place more restrictions on the signers (because of the sequentiality requirement), but relies on a more accepted general assumption.

Finally, we show how to instantiate our construction with the RSA trapdoor permutation. This turns out to be more difficult than may be expected, because in this setting we need to worry about maliciously generated RSA keys (because we need to provide security for User i regardless of whether other users are honest). There are essentially four problems. The first is that our scheme assumes multiple trapdoor permutations over the same domain, which RSA does not provide. The second is that RSA is not a *certified* trapdoor permutation: for a maliciously generated public-key, it can indeed be very far from a permutation. The third is that the domain of RSA is not the convenient $\mathbb{Z}_N$, but rather $\mathbb{Z}_N^*$, which can be much smaller for maliciously generated N. Finally, the natural group operation on $\mathbb{Z}_N^*$ (multiplication) is not a group operation on $\mathbb{Z}_N$. We overcome these problems with techniques that may be of independent interest. In particular, we turn RSA into a *certified* trapdoor permutation over the *entire* $\mathbb{Z}_N$.

Other related work. Aggregate signatures are related to multisignatures [13, 15, 14, 3]. In particular, our aggregate signature scheme has similarities with the multisignature scheme of Okamoto [15] (though the latter has no security proof and, indeed, is missing important details that would make the security proof possible, as shown in [12]). Also of interest are threshold signatures,

[1] A TDP is *certified* [2] if one can verify from the public key that it is actually a permutation.

in particular the non-interactive threshold signature scheme due to Shoup [17], where we have a set of n signers, and a threshold t, such that signature shares from any $t < k \leq n$ signers can be combined into one signature. They are different from aggregate signatures in several crucial aspects: threshold signatures require an expensive (or trusted) setup procedure; pieces of a threshold signature do not constitute a stand-alone signature; pieces of a threshold signature can only be combined into one once there are enough of them; and a threshold signature looks the same no matter which of the signers contributed pieces to it.

2 Preliminaries

We recall the definitions of trapdoor permutations and ordinary digital signatures, and the full-domain hash signatures based on trapdoor permutations. We also define certified trapdoor permutations, which are needed for building sequential aggregate signatures. In addition, we define claw-free permutations, and homomorphic trapdoor permutations, whose properties are used to achieve a better security reduction.

2.1 Trapdoor one-way permutations

Let D be a group over some operation $\odot$. For simplicity, we assume that choosing an element of D at random, computing $\odot$, and inverting $\odot$ each take unit time.

A trapdoor permutation family Π over D comprises three algorithms: *Generate*, *Evaluate*, and *Invert*. The randomized generation algorithm *Generate* outputs the description s of a permutation along with the corresponding trapdoor t. The evaluation algorithm *Evaluate*, given the permutation description s and a value $x \in D$, outputs $a \in D$, the image of x under the permutation. The inversion algorithm *Invert*, given the permutation description s, the trapdoor t, and a value $a \in D$, outputs the preimage of a under the permutation.

We require that $\mathit{Evaluate}(s, \cdot)$ be a permutation of D for all $(s,t) \overset{\mathrm{R}}{\leftarrow} \mathit{Generate}$, and that $\mathit{Invert}(s, t, \mathit{Evaluate}(s,x)) = x$ hold for all $(s,t) \overset{\mathrm{R}}{\leftarrow} \mathit{Generate}$ and for all $x \in D$. The algorithms *Generate*, *Evaluate*, and *Invert* are assumed to take unit time for simplicity.

Definition 2.1. The advantage of an algorithm $\mathcal{A}$ in inverting a trapdoor permutation family is

$$\mathsf{Adv\,Invert}_{\mathcal{A}} \overset{\mathrm{def}}{=} \Pr\left[x = \mathcal{A}(s, \mathit{Evaluate}(s,x)) : (s,t) \overset{\mathrm{R}}{\leftarrow} \mathit{Generate}, x \overset{\mathrm{R}}{\leftarrow} D\right] .$$

The probability is taken over the coin tosses of *Generate* and of $\mathcal{A}$. An algorithm $\mathcal{A}$ (t,ϵ)-inverts a trapdoor permutation family if $\mathcal{A}$ runs in time at most t and $\mathsf{Adv\,Invert}_{\mathcal{A}}$ is at least ϵ. A trapdoor permutation family is (t,ϵ)-one-way if no algorithm (t,ϵ)-inverts the trapdoor permutation family.

Note that this definition of a trapdoor permutation family requires that there exist multiple trapdoor permutations over the same domain D. Note also that we avoid the use of an infinite sequence of domains D, one for each security parameter, by simply fixing the security parameter and considering concrete security.

When it engenders no ambiguity, we consider the output of the generation algorithm *Generate* as a probability distribution Π on permutations, and write $(\pi, \pi^{-1}) \overset{\mathrm{R}}{\leftarrow} \Pi$; here π is the permutation $\mathit{Evaluate}(s, \cdot)$, and π^{-1} is the inverse permutation $\mathit{Invert}(s, t, \cdot)$.

2.2 Certified trapdoor permutations

The trapdoor permutation families used in sequential aggregation must be certified trapdoor permutation families [2]. A certified trapdoor permutation family is one such that, for any string s, it is easy to determine whether s can have been output by *Generate*, and thereby ensure that $\mathit{Evaluate}(s,\cdot)$ is a permutation. This is important when permutation descriptions s can be generated by malicious parties.

Applying the definitions above to the RSA permutation family requires some care. RSA gives permutations over domains $\mathbb{Z}_N^*$, where each user has a distinct modulus N. Moreover, given just a public key (N, e), certifying that the key describes a permutation is difficult. We consider this further in Section 5.

2.3 Claw-free permutations and homomorphic trapdoor permutations

We now describe two variants of trapdoor permutations: claw-free permutations and homomorphic trapdoor permutations. The features these variants provide are not needed in the description of the sequential aggregate signature scheme, but allow a more efficient security reduction in Theorem 4.3.

A *claw-free* permutation family Π [11] is a trapdoor permutation family with an additional permutation $g : D \to D$, evaluated by algorithm $\mathit{EvaluateG}(s,\cdot)$. (More generally, g can map any domain E onto D as long as the uniform distribution on E induces the uniform distribution on $g(E)$.) We assume that algorithm *EvaluateG* runs in unit time, and choosing an element of E at random also takes unit time, just like above.

Definition 2.2. The advantage of an algorithm $\mathcal{A}$ in finding a claw in a claw-free permutation family is

$$\mathsf{Adv\,Claw}_{\mathcal{A}} \overset{\text{def}}{=} \Pr\left[\begin{array}{r} \mathit{Evaluate}(s,x) = \mathit{EvaluateG}(s,y)\ : \qquad\qquad \\ (s,t) \overset{\text{R}}{\leftarrow} \mathit{Generate}, (x,y) \overset{\text{R}}{\leftarrow} \mathcal{A}(s) \end{array}\right] .$$

The probability is taken over the coin tosses of *Generate* and of $\mathcal{A}$. An algorithm $\mathcal{A}$ (t,ϵ)-breaks a claw-free permutation family if $\mathcal{A}$ runs in time at most t and $\mathsf{Adv\,Claw}_{\mathcal{A}}$ is at least ϵ. A permutation family is (t,ϵ)-claw-free if no algorithm (t,ϵ)-breaks the claw-free permutation family.

When it engenders no ambiguity, we abbreviate $\mathit{EvaluateG}(s,\cdot)$ as $g(\cdot)$, and write $(\pi, \pi^{-1}, g) \overset{\text{R}}{\leftarrow} \Pi$. In this compact notation, a claw is a pair (x, y) such that $\pi(x) = g(y)$.

One obtains from every claw-free permutation family a trapdoor permutation family, simply by ignoring *EvaluateG* [11]. The proof is straightforward. Suppose there exists an algorithm $\mathcal{A}$ that inverts π with nonnegligible probability. One selects $y \overset{\text{R}}{\leftarrow} E$, and provides $\mathcal{A}$ with $z = g(y)$, which is uniformly distributed in D. If $\mathcal{A}$ outputs x such that $x = \pi^{-1}(z)$, then it has uncovered a claw $\pi(x) = g(y)$.

A trapdoor permutation family is *homomorphic* if D is a group with some operation $*$ and if, for all (s,t) generated by *Generate*, the permutation $\pi : D \to D$ induced by $\mathit{Evaluate}(s,\cdot)$ is an automorphism on D with $*$. That is, if $a = \pi(x)$ and $b = \pi(y)$, then $a * b = \pi(x * y)$. The group action $*$ is assumed to be computable in unit time. The operation $*$ can be different from the operation $\odot$ given above; we do not require any particular relationship (e.g., distributivity) between $\odot$ and $*$.

One obtains from every homomorphic trapdoor permutation family a claw-free permutation family [10]. Pick some $z \neq 1 \in D$, and define $g(x) = z * \pi(x)$. (In this case, $E = D$.) Then a claw $\pi(x) = g(y) = z * \pi(y)$ reveals $\pi^{-1}(z) = x * (1/y)$ (where the inverse is with respect to $*$).

2.4 Digital signatures

We review the well-known definition of security for ordinary digital signatures.

Existential unforgeability under a chosen message attack [11] in the random oracle model [1] for a signature scheme (*KeyGen*, *Sign*, and *Verify*) with a random oracle H is defined using the following game between a challenger and an adversary $\mathcal{A}$:

Setup. The challenger runs algorithm *KeyGen* to obtain a public key *PK* and private key *SK*. The adversary $\mathcal{A}$ is given *PK*.

Queries. Proceeding adaptively, $\mathcal{A}$ requests signatures with *PK* on at most q_S messages of his choice $M_1, \ldots, M_{q_s} \in \{0,1\}^*$. The challenger responds to each query with a signature $\sigma_i = \mathit{Sign}(\mathit{SK}, M_i)$. Algorithm $\mathcal{A}$ also adaptively asks for at most q_H queries of the random oracle H.

Output. Eventually, $\mathcal{A}$ outputs a pair (M, σ) and wins the game if (1) M is not any of $M_1, \ldots, M_{q_s}$, and (2) $\mathit{Verify}(\mathit{PK}, M, \sigma) = \texttt{valid}$.

We define $\mathsf{Adv\,Sig}_{\mathcal{A}}$ to be the probability that $\mathcal{A}$ wins in the above game, taken over the coin tosses of *KeyGen* and of $\mathcal{A}$.

Definition 2.3. A forger $\mathcal{A}$ (t, q_H, q_S, ϵ)-breaks a signature scheme if $\mathcal{A}$ runs in time at most t; $\mathcal{A}$ makes at most q_S signature queries and at most q_H queries to the random oracle; and $\mathsf{Adv\,Sig}_{\mathcal{A}}$ is at least ϵ. A signature scheme is (t, q_H, q_S, ϵ)-existentially unforgeable under an adaptive chosen-message attack if no forger (t, q_H, q_S, ϵ)-breaks it.

2.5 Full-domain signatures

We review the full-domain hash signature scheme. The scheme, introduced by Bellare and Rogaway [1], works in any trapdoor one-way permutation family. The more efficient security reduction given by Coron [8] additionally requires that the permutation family be homomorphic. Dodis and Reyzin show that Coron's analysis can be applied for any claw-free permutation family [10]. The scheme makes use of a hash function $H : \{0,1\}^* \to D$, which is modeled as a random oracle. The signature scheme comprises three algorithms: *KeyGen*, *Sign*, and *Verify*.

Key generation. For a particular user, pick random $(s,t) \overset{\mathrm{R}}{\leftarrow} \mathit{Generate}$. The user's public key *PK* is s. The user's private key *SK* is (s,t).

Signing. For a particular user, given the private key (s,t) and a message $M \in \{0,1\}^*$, compute $h \leftarrow H(M)$, where $h \in D$, and $\sigma \leftarrow \mathit{Invert}(s,t,h)$. The signature is $\sigma \in D$.

Verification. Given user's public key s, a message M, and a signature σ, compute $h \leftarrow H(M)$; accept if $h = \mathit{Evaluate}(s, \sigma)$ holds.

The following theorem, due to Coron, shows the security of full-domain signatures under the adaptive chosen message attack in the random oracle model. The terms given in the exact analysis of ϵ and t have been adapted to agree with the accounting employed by Boneh *et al* [6].

Theorem 2.4. *Let Π be a (t', ϵ')-one-way homomorphic trapdoor permutation family. Then the full-domain hash signature scheme on Π is (t, q_H, q_S, ϵ)-secure against existential forgery under an adaptive chosen-message attack (in the random oracle model) for all t and ϵ satisfying*

$$\epsilon \geq e(q_S + 1) \cdot \epsilon' \qquad \text{and} \qquad t \leq t' - 2(q_H + 2q_S) \ .$$

Here e is the base of the natural logarithm.

3 Sequential aggregate signatures

We introduce sequential aggregate signatures and present a security model for them.

3.1 Aggregate and sequential aggregate signatures

Boneh *et al.* [5] present a new signature primitive, aggregate signatures. Aggregate signatures are a generalization of multisignatures [13, 15, 14, 3] wherein signatures by several users on several distinct messages may be combined into an aggregate whose length is the same as that of a single signature. Using an aggregate signature in place of several individual signatures in a protocol yields useful space savings. In an aggregate signature, signatures are first individually generated and then combined into an aggregate.

Sequential aggregate signatures are different. Each would-be signer transforms a sequential aggregate into another that includes a signature on a message of his choice. Signing and aggregation are a single operation; sequential aggregates are built in layers, like an onion; the first signature in the aggregate is the inmost. As with non-sequential aggregate signatures, the resulting sequential aggregate is the same length as an ordinary signature. This behavior closely mirrors the sequential nature of certificate chains in a PKI.

Let us restate the intuition given above more formally. Aggregation and signing is a combined operation in a sequential aggregate signature scheme. The operation takes as input a private key SK, a message M_i to sign, and a sequential aggregate σ' on messages $M_1, \ldots, M_{i-1}$ under respective public keys $PK_1, \ldots, PK_{i-1}$, where M_1 is the inmost message. All of $M_1, \ldots, M_{i-1}$ and $PK_1, \ldots, PK_{i-1}$ must be provided as inputs. If i is 1, the aggregate σ is taken to be empty. It adds a signature on M_i under SK to the aggregate, outputting a sequential aggregate σ on all i messages $M_1, \ldots, M_i$.

The aggregate verification algorithm is given a sequential aggregate signature σ, messages $M_1, \ldots, M_i$, and public keys $PK_1, \ldots, PK_i$, and verifies that σ is a valid sequential aggregate (with M_1 inmost) on the given messages under the given keys.

3.2 Sequential aggregate signature security

The security of sequential aggregate signature schemes is defined as the nonexistence of an adversary capable, within the confines of a certain game, of existentially forging a sequential aggregate signature. Existential forgery here means that the adversary attempts to forge a sequential aggregate signature, on messages of his choice, by some set of users not all of whose private keys are known to the forger.

We formalize this intuition as the sequential aggregate chosen-key security model. In this model, the adversary $\mathcal{A}$ is given a single public key. His goal is the existential forgery of a sequential aggregate signature. We give the adversary power to choose all public keys except the challenge public key. The adversary is also given access to a sequential aggregate signing oracle on the challenge key. His advantage, $\mathsf{Adv\,AggSig}_{\mathcal{A}}$, is defined to be his probability of success in the following game.

Setup. The aggregate forger $\mathcal{A}$ is provided with a public key PK, generated at random.

Queries. Proceeding adaptively, $\mathcal{A}$ requests sequential aggregate signatures with PK on messages of his choice. For each query, he supplies a sequential aggregate signature σ on some messages $M_1, \ldots, M_{i-1}$ under distinct keys $PK_1, \ldots, PK_{i-1}$, and an additional message M_i to be signed by the oracle under key PK (where i is at most n, a game parameter).

Response. Finally, $\mathcal{A}$ outputs i distinct public keys $PK_1, \ldots, PK_i$. Here i is at most n, and need not equal the lengths (also denoted i) of $\mathcal{A}$'s requests in the query phase above. One of these keys must equal PK, the challenge key. Algorithm $\mathcal{A}$ also outputs messages $M_1, \ldots, M_i$, and a sequential aggregate signature σ by the i users, each on his corresponding message, with PK_1 inmost.

The forger wins if the sequential aggregate signature σ is a valid sequential aggregate signature on messages $M_1, \ldots, M_i$ under keys $PK_1, \ldots, PK_i$, and σ is nontrivial, i.e., $\mathcal{A}$ did not request a sequential aggregate signature on messages $M_1, \ldots, M_{i^*}$ under keys $PK_1, \ldots, PK_{i^*}$, where i^* is the index of the challenge key PK in the forgery. The probability is over the coin tosses of the key-generation algorithm and of $\mathcal{A}$.

Definition 3.1. A sequential aggregate forger $\mathcal{A}$ $(t, q_H, q_S, n, \epsilon)$-breaks an n-user aggregate signature scheme in the sequential aggregate chosen-key model if: $\mathcal{A}$ runs in time at most t; $\mathcal{A}$ makes at most q_H queries to the hash function and at most q_S queries to the aggregate signing oracle; $\mathsf{Adv\,AggSig}_{\mathcal{A}}$ is at least ϵ; and the forged sequential aggregate signature is by at most n users. A sequential aggregate signature scheme is $(t, q_H, q_S, n, \epsilon)$-secure against existential forgery in the sequential aggregate chosen-key model if no forger $(t, q_H, q_S, n, \epsilon)$-breaks it.

4 Sequential aggregates from trapdoor permutations

We describe a sequential aggregate signature scheme arising from any family of trapdoor permutations, and prove the security of the scheme.

We first introduce some notation for vectors. We write a vector as $\mathbf{x}$, its length as $|\mathbf{x}|$, and its elements as $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|\mathbf{x}|}$. We denote concatenating vectors as $\mathbf{x}\|\mathbf{y}$ and appending an element to a vector as $\mathbf{x}\|z$. For a vector $\mathbf{x}$, $\mathbf{x}|_a^b$ is the sub-vector containing elements $\mathbf{x}_a, \mathbf{x}_{a+1}, \ldots, \mathbf{x}_b$. It is necessarily the case that $1 \le a \le b \le |\mathbf{x}|$.

4.1 The scheme

We now describe three algorithms: *KeyGen*, *AggregateSign*, and *AggregateVerify* for our sequential aggregate signature scheme. The scheme employs a full-domain hash function $H : \{0,1\}^* \to D$, viewed as a random oracle, and resembles full-domain hash described in Section 2.5. The trick to aggregation is to incorporate the sequential aggregate signature of previous users by multiplying it (via the group operation $\odot$) together with the hash of the message. Actually, the hash now needs to include not only the signer's message, but also her public key and the prior messages and keys.[2]

Key generation. For a particular user, pick random $(s, t) \xleftarrow{\text{R}} \mathit{Generate}$. The user's public key PK is s. The user's private key SK is (s, t).

Aggregate signing. The input is a private key (s, t), a message $M \in \{0, 1\}^*$ to be signed, and a sequential aggregate σ' on messages $\mathbf{M}$ under public keys $\mathbf{s}$. Verify that σ' is a valid signature on $\mathbf{M}$ under $\mathbf{s}$ using the verification algorithm below; if not, output $\star$, indicating error. Otherwise, compute $h \leftarrow H(\mathbf{s}\|s, \mathbf{M}\|M)$, where $h \in D$, and $\sigma \leftarrow \mathit{Invert}(s, t, h \odot \sigma')$. The sequential aggregate signature is $\sigma \in D$.

Aggregate verification. The input is a sequential aggregate σ on messages $\mathbf{M}$ under public keys $\mathbf{s}$. If any key appears twice in $\mathbf{s}$, if any element of $\mathbf{s}$ does not describe a valid permutation, or if $|\mathbf{M}|$ and $|\mathbf{s}|$ differ, reject. Otherwise, let i equal $|\mathbf{M}| = |\mathbf{s}|$. Set $\sigma_i \leftarrow \sigma$. Then, for $j = i, \ldots, 1$, set $\sigma_{j-1} \leftarrow \mathit{Evaluate}(\mathbf{s}_j, \sigma_j) \odot H(\mathbf{s}|_1^j, \mathbf{M}|_1^j)^{-1}$. Accept if σ_0 equals 1, the unit of D with respect to $\odot$.

Written using π-notation, a sequential aggregate signature is of the form

$$\pi_i^{-1}(h_i \odot \pi_{i-1}^{-1}(h_{i-1} \odot \pi_{i-2}^{-1}(\cdots \pi_2^{-1}(h_2 \odot \pi_1^{-1}(h_1))\cdots))) \ ,$$

where $h_j = H(\mathbf{s}|_1^j, \mathbf{M}|_1^j)$. Verification evaluates the permutations in the forward direction, peeling layers away until the center is reached.

4.2 Security

The following theorem demonstrates that our is secure when instantiated on any certified trapdoor permutation family.

Theorem 4.1. *Let Π be a certified (t', ϵ')-trapdoor permutation family. Then our sequential aggregate signature scheme on Π is $(t, q_H, q_S, n, \epsilon)$-secure against existential forgery under an adaptive sequential aggregate chosen-message attack (in the random oracle model) for all t and ϵ satisfying*

$$\epsilon \geq (q_H + q_S + 1) \cdot \epsilon' \qquad \text{and} \qquad t \leq t' - (4nq_H + 4nq_S + 7n - 1) \ .$$

Following to Coron's work [8], a better security reduction is obtained if the trapdoor permutations are, additionally, homomorphic under some operatin $*$. (The operation $*$ need not be the same as the operation $\odot$ used in the description of the signature scheme in Section 4.)

Theorem 4.2. *Let Π be a certifified homomorphic (t', ϵ')-trapdoor permutation family. Then our sequential aggregate signature scheme on Π is $(t, q_H, q_S, n, \epsilon)$-secure against existential forgery under an adaptive sequential aggregate chosen-message attack (in the random oracle model) for all t and ϵ satisfying*

$$\epsilon \geq e(q_S + 1) \cdot \epsilon' \qquad \text{and} \qquad t \leq t' - ((4n + 1)q_H + (4n + 1)q_S + 7n + 3) \ .$$

Here e is the base of the natural logarithm.

[2] This is done not merely because we do not know how to prove the scheme secure otherwise. Micali *et al.* [13] pointed out that if the signature does not include the public key, then an adversary may attack the scheme by deciding on the public key after the signature is issued. Our approach is the same as that of Boneh *et al.* [5, Section 3.2].

Finally, following the work of Dodis and Reyzin [10], the homorphic property is not really necessary, and can be replaced with the more general claw-free property:

Theorem 4.3. *Let* Π *be a certified* (t', ϵ')*-claw-free permutation family. Then the sequential aggregate signature scheme on* Π *is* $(t, q_H, q_S, n, \epsilon)$*-secure against existential forgery under an adaptive sequential aggregate chosen-message attack (in the random oracle model) for all* t *and* ϵ *satisfying*

$$\epsilon \geq e(q_S + 1) \cdot \epsilon' \qquad \textit{and} \qquad t \leq t' - (4nq_H + 4nq_S + 7n) \enspace .$$

Here e *is the base of the natural logarithm.*

The proofs of these theorems are very similar (in fact, Theorem 4.2 is just a corollary of Theorem 4.3, because, as we already saw, homomorphic trapdoor permutations are claw-free). We will prove all three at once.

Proofs. Suppose there exists a forger $\mathcal{A}$ that breaks the security of our sequential aggregate signature scheme. We demonstrate algorithm $\mathcal{B}$ that reduces $\mathcal{A}$ to breaks of the different security notions (trapdoor one-wayness, homomorphic one-wayness, and claw-freeness). In fact, the algorithm is slightly different depending on the security assumption that it is trying to break. The variant we present uses $\mathcal{A}$ to find a claw in a (supposedly) claw-free permutation family Π. We will point out later the changes needed to make the reduction to ordinary and homormophic trapdoor permutations.

Suppose $\mathcal{A}$ is a forger algorithm that $(t, q_H, q_S, n, \epsilon)$-breaks the sequential aggregate signature scheme. We construct an algorithm $\mathcal{B}$ that finds a claw in Π.

Crucial in our construction is the following fact about our signature scheme: once the function H is fixed on i input values $(\mathbf{s}|_1^j, \mathbf{M}|_1^j)$, $1 \leq j \leq i$, there exists only one valid aggregate signature on $\mathbf{M}$ using keys $\mathbf{s}$. Thus, by answering hash queries properly, $\mathcal{B}$ can prepare for answering signature queries and for taking advantage of the eventual forgery.

Algorithm $\mathcal{B}$ is given the description s of an element of Π, and must find values $x \in D$ and $y \in E$ such that $\mathit{Evaluate}(s, x) = \mathit{EvaluateG}(s, y)$. Algorithm $\mathcal{B}$ supplies $\mathcal{A}$ with the public key s. It then runs $\mathcal{A}$ and answers its oracle queries as follows.

Hash queries. Algorithm $\mathcal{B}$ maintains a list of tuples $\langle \mathbf{s}^{(j)}, \mathbf{M}^{(j)}, w^{(j)}, r^{(j)}, c^{(j)} \rangle$. We refer to this list as the H-list. The list is initially empty. When $\mathcal{A}$ queries the oracle H at a point $(\mathbf{s}, \mathbf{M})$, algorithm $\mathcal{B}$ responds as follows.

If the query $(\mathbf{s}, \mathbf{M})$ is already present on the H-list, in some tuple $\langle \mathbf{s}, \mathbf{M}, w, r, c \rangle$, algorithm $\mathcal{B}$ answers the query as $H(\mathbf{s}, \mathbf{M}) = w \in D$.

If $|\mathbf{M}|$ and $|\mathbf{s}|$ differ, if $|\mathbf{s}|$ exceeds n, if some key is repeated in $\mathbf{s}$, or if any key in $\mathbf{s}$ does not describe a valid permutation, then $(\mathbf{s}, \mathbf{M})$ can never be part of a sequential aggregate signature. Algorithm $\mathcal{B}$ picks $w \xleftarrow{\mathrm{R}} D$, and sets $r \leftarrow \star$ and $c \leftarrow \star$, both placeholder values. It adds $\langle \mathbf{s}, \mathbf{M}, w, r, c \rangle$ to the H-list and responds to the query as $H(\mathbf{s}, \mathbf{M}) = w \in D$.

Set $i = |\mathbf{s}| = |\mathbf{M}|$. If i exceeds 1, $\mathcal{B}$ runs the hashing algorithm on input $(\mathbf{s}|_1^{i-1}, \mathbf{M}|_1^{i-1})$, obtaining the corresponding entry on the H-list, $\left\langle \mathbf{s}|_1^{i-1}, \mathbf{M}|_1^{i-1}, w', r', c' \right\rangle$. If i equals 1, $\mathcal{B}$ sets $r' \leftarrow 1$. Algorithm $\mathcal{B}$ must now choose elements r,w, and c to include, along with $\mathbf{s}$ and $\mathbf{M}$, in a new entry on the H-list. There are three cases to consider.

If the challenge key s does not appear at any index of $\mathbf{s}$, $\mathcal{B}$ chooses $r \xleftarrow{\mathrm{R}} D$ at random, sets $c \leftarrow \star$, a placeholder value, and computes

$$w \leftarrow \mathit{Evaluate}(\mathbf{s}_i, r) \odot (r')^{-1} \enspace .$$

If the challenge key s appears in $\mathbf{s}$ at index $i^* = i$, Algorithm $\mathcal{B}$ generates a random coin $c \in \{0, 1\}$ such that $\Pr[c = 0] = 1/(q_S + 1)$. If $c = 1$, $\mathcal{B}$ chooses $r \xleftarrow{\mathrm{R}} D$ at random and sets

$$w \leftarrow \mathit{Evaluate}(s, r) \odot (r')^{-1} \enspace .$$

(In this case, w is uniform in D and independent of all other queries because r has been chosen uniformly and independently at random from D, and *Evaluate* and combining with $(r')^{-1}$ are both permutations.) If $c = 0$, $\mathcal{B}$ chooses $r \xleftarrow{\mathrm{R}} E$ at random and sets

$$w \leftarrow \mathit{EvaluateG}(s, r) \odot (r')^{-1} \enspace .$$

(In this case, w is uniform in D and independent of all other queries because r has been chosen uniformly and independently at random from E, *EvaluateG* maps uniformly onto D, and combining with $(r')^{-1}$ is a permutation.)

If the challenge key s appears in $\mathbf{s}$ at index $i^* \lneq i$, aggregate signature. $\mathcal{B}$ picks $w \xleftarrow{\mathrm{R}} D$ at random, and sets $r \leftarrow \star$ and $c \leftarrow \star$, both placeholder values.

Finally, $\mathcal{B}$ adds $\langle \mathbf{s}, \mathbf{M}, w, r, c \rangle$ to the H-list, and responds to the query as $H(\mathbf{s}, \mathbf{M}) = w$.

In all cases, $\mathcal{B}$'s response, w, is uniform in D and independent of $\mathcal{A}$'s current view, as required.

Aggregate signature queries. Algorithm $\mathcal{A}$ requests a sequential aggregate signature, under key s, on messages $\mathbf{M}$ under keys $\mathbf{s}$.

If $|\mathbf{s}|$ and $|\mathbf{M}|$ differ, if $|\mathbf{s}|$ exceeds n, if any key appears more than once in $\mathbf{s}$, or if any key in $\mathbf{s}$ does not describe a valid permutation, $(\mathbf{s}, \mathbf{M})$ is not a valid aggregate, and $\mathcal{B}$ responds to $\mathcal{A}$ with $\star$, indicating error. Let $i = |\mathbf{s}| = |\mathbf{M}|$. If $\mathbf{s}_i$ differs from s, $(\mathbf{s}, \mathbf{M})$ is not a valid query to the aggregate signing oracle, and $\mathcal{B}$ again responds with $\star$.

Algorithm $\mathcal{A}$ also supplies a purported aggregate signature σ' on messages $\mathbf{M}|_1^{i-1}$ under keys $\mathbf{s}|_1^{i-1}$. If i equals 1, $\mathcal{B}$ verifies that σ' equals 1. Otherwise, $\mathcal{B}$ uses *AggregateVerify* to ensure that σ' is the correct sequential aggregate signature on $(\mathbf{s}|_1^{i-1}, \mathbf{M}|_1^{i-1})$. If σ' is incorrect, $\mathcal{B}$ again responds with $\star$.

Otherwise, $\mathcal{B}$ runs the hash algorithm on $(\mathbf{s}, \mathbf{M})$, obtaining the corresponding entry on the H-list, $\langle \mathbf{s}, \mathbf{M}, w, r, c \rangle$. Since $\mathbf{s}_i$ equals s, c must be 0 or 1. If $c = 0$ holds, $\mathcal{B}$ reports failure and terminates. Otherwise, $\mathcal{B}$ responds to the query with $\sigma \leftarrow r$.

Output. Eventually algorithm $\mathcal{A}$ halts, producing a message vector $\mathbf{M}$, a public-key vector $\mathbf{s}$, and a corresponding sequential aggregate signature forgery σ. The forgery must be valid: No key may occur more than once in $\mathbf{s}$, each key in $\mathbf{s}$ must describe a valid permutation, the two vectors $\mathbf{s}$ and $\mathbf{M}$ must have the same length i, which is at most n. The forgery must also be nontrivial: The challenge key s must occur in $\mathbf{s}$, at some location i^*, and $\mathcal{A}$ must not have asked for a sequential aggregate signature on messages $\mathbf{M}|_1^{i^*}$ under keys $\mathbf{s}|_1^{i^*}$. If $\mathcal{A}$ fails to output a valid and nontrivial forgery, $\mathcal{B}$ reports failure and terminates.

Algorithm $\mathcal{B}$ begins by checking the hashes included in σ. For each j, $1 \le j \le i$, $\mathcal{B}$ runs its hash algorithm on $(\mathbf{s}|_1^j, \mathbf{M}|_1^j)$, obtaining a series of tuples $\left\langle \mathbf{s}|_1^j, \mathbf{M}|_1^j, w^{(j)}, r^{(j)}, c^{(j)} \right\rangle$. Note that $\mathcal{B}$ always returns w as the answer to a hash query, so, for each j, $H(\mathbf{s}|_1^j, \mathbf{M}|_1^j) = w^{(j)}$.

Algorithm $\mathcal{B}$ then examines $c^{(i^*)}$. Since $s^{(i^*)}$ equals s, $c^{(i^*)}$ must be 0 or 1. If $c^{(i^*)} = 1$ holds, $\mathcal{B}$ reports failure and terminates. Then $\mathcal{B}$ applies the aggregate signature verification algorithm to σ. It sets $\sigma^{(i)} \leftarrow \sigma$. For $j = i, \dots, 1$, it sets $\sigma^{(j-1)} \leftarrow \mathit{Evaluate}(s^{(j)}, \sigma^{(j)}) \odot (w^{(j)})^{-1}$.

If $\sigma^{(0)}$ does not equal 0, σ is not a valid aggregate signature, and $\mathcal{B}$ reports failure and terminates. Otherwise, σ is valid and, moreover, each $\sigma^{(j)}$ computed by $\mathcal{B}$ is the (unique) valid aggregate signature on messages $\mathbf{M}|_1^j$ under keys $\mathbf{s}|_1^j$.

Finally, $\mathcal{B}$ sets $x \leftarrow \sigma^{(i^*)}$ and $y \leftarrow r^{(i^*)}$.

This completes the description of algorithm $\mathcal{B}$.

It is easy to modify this algorithm for homomorphic trapdoor permutations. Now the algorithm's goal is not to find a claw, but to invert the permutation given by s on a given input z. Simply replace, when answering hash queries for $c = 0$, invocation of $\mathit{EvaluateG}(s, r)$ with $z * \mathit{Evaluate}(s, r)$. The a claw (x, y) allows $\mathcal{B}$ to recover the inverse of z under the permutation by computing $z = x * (1/y)$, where $1/y$ is the inverse of y under $*$.

Finally, it is also easy to modify this algorithm for ordinary trapdoor permutations:

- In answering hash queries where the challenge key s is outmost in $\mathbf{s}$, instead of letting $c = 0$ with probability $1/(q_S + 1)$, set $c = 0$ for exactly one query, chosen at random. There can be at most $q_H + q_S + 1$ such queries.
- For the $c = 0$ query, set $w \leftarrow z \odot (r')^{-1}$. Then w is random given $\mathcal{A}$'s view.
- If Algorithm $\mathcal{A}$'s forgery is such that $c^{(i^*)} = 0$, $\mathcal{B}''$ outputs $x \leftarrow \sigma^{(i^*)}$.

In Appendix A we show that $\mathcal{B}$ correctly simulates $\mathcal{A}$'s environment, and analyze its running time and success probability. □

5 Aggregating with RSA

We consider the details of instantiating the sequential aggregate signature scheme presented in Section 4 using the RSA permutation family.

The RSA function was introduced by Rivest, Shamir, and Adleman [16]. If $N = pq$ is the product of two large primes and $ed = 1 \bmod \phi(N)$, then $\pi(x) = x^e \bmod N$ is a permutation on $\mathbb{Z}_N^*$, and $\pi^{-1}(x) = x^d \bmod N$ is its inverse. Setting $s = (N, e)$ and $t = (d)$ gives a one-way trapdoor permutation that is multiplicatively homomorphic.

A few difficulties arise when we try to instantiate the above scheme with RSA. We tackle them individually.

The first problem is that RSA is not a *certified* trapdoor permutation. Raising to the power e may not be a permutation over $\mathbb{Z}_N^*$ if e is not relatively prime with $\phi(N)$. Moreover, even if it is a permutation of $\mathbb{Z}_N^*$, it may not be a permutation of the entire $\mathbb{Z}_N$ if N is maliciously generated (in particular, if N is not square-free). Note that, for maliciously generated N, the difference between $\mathbb{Z}_N^*$ and $\mathbb{Z}_N$ may be considerable. The traditional argument used to dismiss this issue (that if one finds x outside $\mathbb{Z}_N^*$, one factors N) has no relevance here: N may be generated by the adversary, and our ability to factor it has no impact on the security of the scheme for the honest signer who is using a different modulus. Our security proof substantially relied on the fact that even the adversarial public keys define permutations, for uniqueness of signatures and proper distribution of hash query answers. Indeed, this is not just a "proof problem," but a demonstrable security concern: If the adversary is able to precede the honest user's key (N_i, e_i) with multiple keys $(N_1, e_1), \dots, (N_{i-1}, e_{i-1})$, each of which defines a collision-prone function rather than a permutation, then it is quite possible that no matter value one takes for σ_i, it will be likely to verify correctly: for example, there will be two valid σ_1 values, four valid σ_2 values, eight valid σ_3 values, $\dots$, 2^i valid σ_i values.

We tackle this problem in the same way as Micali *et al.* [12]. First, we require e to be a prime larger than N (this idea also appeared in Cachin *et al.* [7]). Then it is guaranteed to be relatively prime with $\phi(N)$, and therefore provide a permutation over $\mathbb{Z}_N^*$. To extend to a permutation over $\mathbb{Z}_N$, we define *Evaluate*$((N, e), x)$ as follows: if $\gcd(x, N) = 1$, output $x^e \bmod N$; else output x.

The second problem is that the natural choice for the group operation $\odot$, multiplication, is not actually a group operation over $\mathbb{Z}_N$. Thus, signature verification, which requires computation of an inverse under $\odot$, may be unable to proceed. Moreover, our security proof, which relies on the fact that $\odot$ is a group operation for uniqueness of signatures and proper distribution of hash query answers, will no longer hold. This difficulty is simple to overcome: Use addition modulo N as the group operation $\odot$. Recall that no properties were required of $\odot$ beyond being a group operation on the domain.

The third problem is that two users cannot share the same modulus N. Thus the domains of the one-way permutations belonging to the aggregating users differ, making it difficult to treat RSA as a family of trapdoor permutations. We give two approaches that allow us to create sequential aggregates from RSA nonetheless.

The first approach is to require the users' moduli to be arranged in increasing order: $N_1 < N_2 \ldots < N_n$. At verification, it is important to check that the i-th signature σ_i is actually less than N_i, to ensure that correct signatures are unique if H is fixed. As long as $\log N_1 - \log N_n$ is constant, and the range of H is a subset of $\mathbb{Z}_{N_1}$ whose size is a constant fraction of N_1, the scheme will be secure. The same security proof still goes through, with the following minor modification for answering hash queries. Whenever a hash query answer w is computed by first choosing a random r in $\mathbb{Z}_{N_i}$, there is a chance that w will be outside of the range of H. In this case, simply repeat with a fresh random r until w falls in the right range (the expected number of repetitions is constant). Note that because we insisted on *Evaluate* being a permutation and $\odot$ being a group operation, the resulting distribution of w is uniform on the range of H. Therefore, the distribution of answers to hash queries is uniform. Since signatures are uniquely determined by answers to hash queries, the adversary's whole view is correct, and the proof works without other modifications. (This technique is related to Coron's partial-domain hash analysis [9], though Coron deals with the more complicated case when the partial domain is exponentially smaller than the full domain.)

Our second approach allows for more general moduli: we do not require them to be in increasing order. However, we do require them to be of the same length l (constant differences in the lengths will also work, but we do not address them here for simplicity of exposition). The signature will expand by n bits $b_1 \ldots b_n$, where n is the total number of users. Namely, during signing, if $\sigma_i \geq N_{i+1}$, let $b_i = 1$; else, let $b_i = 0$. During verification, if $b_i = 1$, add N_{i+1} to σ_i before proceeding with the verification of σ_i. Always check that σ_i is in the correct range $0 \leq \sigma_i < N_i$ (to ensure, again, uniqueness of signatures). The security proof requires no major modifications.[3]

To summarize, the resulting RSA aggregate signature schemes for n users with moduli of length l are as follows. Let $H : \{0,1\}^* \to \{0,1\}^{l-1}$ be a hash function.

Restricted Moduli. We first present the scheme where the moduli must be ordered.

Key generation. Each user i generates an RSA public key (N_i, e_i) and secret key (N_i, d_i), ensuring that $2^{l-1}(1 + (i-1)/n) \leq N_i < 2^{l-1}(1 + i/n)$ and that $e_i > N_i$ is a prime.

Signing. First, given an aggregate signature σ' on $M_1, \ldots, M_{i-1}$ and keys $(N_1, e_1), \ldots, (N_{i-1}, e_{i-1})$, user i first verifies σ' (using the verification procedure below). If the verification succeeds, user i computes $h_i = H((M_1, \ldots, M_i), ((N_1, e_1), \ldots, (N_i, e_i)))$, $y = h_i + \sigma'$ and outputs $\sigma = y^{d_i} \bmod N_i$ (the user may first check that $\gcd(y', N) = 1$ and, if not, output y; however, the chances that the check will fail are negligible, because the user is honest).

Verifying. On input an aggregate signature σ on $M_1, \ldots, M_i$ and keys $(N_1, e_1), \ldots, (N_{i-1}, e_i)$, first check that no key appears twice. Then check that $e_i > N_i$ is a prime and that N_i is of length l (this can be once per key, and need not be done per signature). Check that $0 \leq \sigma < N_i$. If $\gcd(\sigma, N_i) = 1$, let $y = \sigma^{e_i} \bmod N_i$. Else let $y = \sigma$ (this check is crucial, because we do not know if user i is honest). Compute $h_i = H((M_1, \ldots, M_i), ((N_1, e_1), \ldots, (N_i, e_i)))$ and $\sigma' = y - h_i \bmod N_i$. Verify σ' recursively. The base case for recursion is $i = 0$, in which case simply check that $\sigma = 0$.

Unrestricted Moduli. We present the scheme for unordered moduli by simply demonstrating the required modifications. First, the range of N_i is now $2^{l-1} < N_i < 2^l$. Second, to sign, upon verifying σ', check if $\sigma' \geq N_i$. If so, replace σ' with $\sigma' - N_i$ and set $b_i = 1$; else, set $b_i = 0$. Finally, to verify, replace σ' with $\sigma' + b_i N_i$ before proceeding with the recursive step.

Security. Because RSA over $\mathbb{Z}_N^*$ is homomorphic with respect to multiplication, it is claw-free (not just over $\mathbb{Z}_N^*$, but over entire $\mathbb{Z}_N$, because finding a claw outside of $\mathbb{Z}_n^*$ implies factoring n and hence being able to invert RSA). Therefore, the conclusions of Theorem 4.3 apply to this scheme.

Acknowledgments

The authors thank Dan Boneh, Stanisław Jarecki, and Craig Gentry for helpful discussions about this work, and Eu-Jin Goh for his detailed and helpful comments on the manuscript.

References

[1] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In D. Denning, R. Pyle, R. Ganesan, R. Sandhu, and V. Ashby, editors, *Proceedings of CCS 1993*, pages 62–73. ACM, 1993.

[3] We need to argue that correct signatures are unique given the hash answers. At first glance it may seem that the adversary may have choice on whether to use $b_i = 0$ or $b_i = 1$. However, this will result in two values σ_{i-1} that are guaranteed to be different: one will be less than N_i and the other at least N_i. Hence uniqueness of σ_{i-1} implies uniqueness of b_i and, therefore, σ_i. Thus, by induction, signatures are still unique. In particular, there is no need to include b_i into the hash function input.

[2] M. Bellare and M. Yung. Certifying permutations: Non-interactive zero-knowledge based on any trapdoor permutation. *J. Cryptology*, 9(1):149–66, 1996.

[3] A. Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-Diffie-Hellman-group signature scheme. In Y. Desmedt, editor, *Proceedings of PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer-Verlag, 2003.

[4] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. Computing*, 32(3):586–615, 2003. Extended abstract in *Proceedings of Crypto 2001.*

[5] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In E. Biham, editor, *Proceedings of Eurocrypt 2003*, volume 2656 of *LNCS*, pages 416–32. Springer-Verlag, 2003.

[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Proceedings of Asiacrypt 2001*, volume 2248 of *LNCS*, pages 514–32. Springer-Verlag, 2001. Full paper: `http://crypto.stanford.edu/~dabo/pubs.html`.

[7] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In J. Stern, editor, *Proceedings of Eurocrypt 1999*, volume 1592 of *LNCS*, pages 402–414. Springer-Verlag, 1999.

[8] J.-S. Coron. On the exact security of full domain hash. In M. Bellare, editor, *Proceedings of Crypto 2000*, volume 1880 of *LNCS*, pages 229–35. Springer-Verlag, 2000.

[9] J.-S. Coron. Security proof for partial-domain hash signature schemes. In M. Yung, editor, *Proceedings of Crypto 2002*, volume 2442 of *LNCS*, pages 613–26. Springer-Verlag, 2002.

[10] Y. Dodis and L. Reyzin. On the power of claw-free permutations. In S. Cimato, C. Galdi, and G. Persiano, editors, *Proceedings of SCN 2002*, number 2576 in LNCS, pages 55–73. Springer-Verlag, 2002.

[11] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, 1988.

[12] S. Micali, K. Ohta, and L. Reyzin. Provable-subgroup signatures. Unpublished manuscript, 1999.

[13] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures (extended abstract). In *Proceedings of CCS 2001*, pages 245–54. ACM Press, 2001.

[14] K. Ohta and T. Okamoto. Multisignature schemes secure against active insider attacks. *IEICE Trans. Fundamentals*, E82-A(1):21–31, 1999.

[15] T. Okamoto. A digital multisignature scheme using bijective public-key cryptosystems. *ACM Trans. Computer Systems*, 6(4):432–41, November 1988.

[16] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Commun. ACM*, 21:120–126, 1978.

[17] V. Shoup. Practical threshold signatures. In B. Preneel, editor, *Proceedings of Eurocrypt 2000*, volume 1807 of *LNCS*, pages 207–220. Springer Verlag, 2000.

A Analysis of the security reduction

In this seciton we analyze the success probability of the reduction $\mathcal{B}$ described in in the proofs of Theorems 4.1, 4.2, and 4.3.

Recall that we suppose that $\mathcal{A}$ is a forger algorithm that $(t, q_H, q_S, n, \epsilon)$-breaks the sequential aggregate signature scheme. Our goal is to show that algorithm $\mathcal{B}$, described above, correctly simulates $\mathcal{A}$'s environment, runs in time t', and finds a claw in Π with probability at least ϵ', which will contradict the (t', ϵ')-claw-freeness of Π (almost identical arguments work for the modifications of $\mathcal{B}$ for ordinary and homomorphic trapdoor permutations).

We introduce some notation, which we will use to demonstrate that $\mathcal{B}$ correctly answers $\mathcal{A}$'s oracle queries. Consider public keys $\mathbf{s}$ and respective messages $\mathbf{M}$, where $i = |\mathbf{s}| = |\mathbf{M}|$, and the entries in $\mathbf{s}$ are all distinct. For each j, $1 \le j \le i$, $\mathcal{B}$'s hash algorithm associates with $(\mathbf{s}|_1^j, \mathbf{M}|_1^j)$ a tuple $\left\langle \mathbf{s}|_1^j, \mathbf{M}|_1^j, w^{(j)}, r^{(j)}, c^{(j)} \right\rangle$. The last three elements of these tuples we view as i-element vectors $\mathbf{w}$, $\mathbf{r}$, and $\mathbf{c}$. Algorithm $\mathcal{B}$ always returns w as the answer to a hash query, so, for each j, $H(\mathbf{s}|_1^j, \mathbf{M}|_1^j) = \mathbf{w}_j$. We also abbreviate the permutation evaluation $\mathit{Evaluate}(\mathbf{s}_j, \cdot)$ as $\pi_j(\cdot)$. For each j there is a unique correct sequential aggregate signature σ_j on messages $\mathbf{M}|_1^j$ under keys $\mathbf{s}|_1^j$. Finally, for the challenge key s, we abbreviate the second function of the claw-free permutation pair, $\mathit{EvaluateG}(s, \cdot)$, as $g(\cdot)$.

Note that, for the keys $\mathbf{s}$ and messages $\mathbf{M}$ output by $\mathcal{A}$ as its forgery, $\mathcal{B}$, in its output phase, computes and makes use of the vectors $\mathbf{w}$, $\mathbf{r}$, and $\mathbf{c}$ as defined here, along with the correct sequential aggregate signatures σ_j. In our analysis, we will also consider these vectors for keys and messages other than those forged on by $\mathcal{A}$.

The proof proceeds in a series of claims. In particular, Claim 4 below shows that $\mathcal{B}$ answers $\mathcal{A}$'s signature queries with the correct sequential aggregate signature, and Claim 5 below shows that $\mathcal{B}$ outputs a claw $\pi(x) = g(y)$.

Claim 1. *If the challenge key s does not equal any of the elements of $\mathbf{s}$, then $\sigma_j = \mathbf{r}_j$ for each j, $1 \le j \le i$.*

Proof. We proceed by induction. Since $\mathbf{s}_1 \ne s$, $\mathbf{w}_1 = \pi_1(\mathbf{r}_1) \odot 1$, or, equivalently, $\mathbf{r}_1 = \pi_1^{-1}(\mathbf{w}_1) = \pi_1^{-1}(H(\mathbf{s}|_1^1, \mathbf{M}|_1^1)) = \sigma_1$. Thus the claim holds for $j = 1$. If the claim holds for $j - 1$, then, since $\mathbf{s}_j \ne s$, $\mathbf{w}_j = \pi_j(\mathbf{r}_j) \odot {\mathbf{r}_{j-1}}^{-1}$, or, equivalently, $\mathbf{r}_j = \pi_j^{-1}(\mathbf{w}_j \odot \mathbf{r}_{j-1}) = \pi_1^{-1}(H(\mathbf{s}|_1^j, \mathbf{M}|_1^j) \odot \sigma_{j-1}) = \sigma_j$, and the claim holds for j. □

Claim 2. *If the challenge key s appears at index i^* of $\mathbf{s}$, and $\mathbf{c}_{i^*} = 1$, then $\sigma_j = \mathbf{r}_j$ for each j, $1 \le j \le i^*$.*

Proof. If $\mathbf{c}_{i^*}$ equals 1, then $\mathcal{B}$ computes $\mathbf{w}_{i^*}$ precisely as it would have had $\mathbf{s}_{i^*}$ not been s. Thus the proof of Claim 1 applies still. □

Claim 3. *If the challenge key s appears at index i^* of $\mathbf{s}$, and $\mathbf{c}_{i^*} = 0$, then, for $j < i^*$, $\sigma_j = \mathbf{r}_j$, and, for $j = i^*$, $\sigma_j = \pi_{i^*}^{-1}(g(\mathbf{r}_{i^*}))$.*

Proof. For $j < i^*$, the result follows from Claim 1. We consider the case $j = i^*$. If i^* equals 1, $\mathcal{B}$ calculates the hash $\mathbf{w}_{i^*}$ as

$$\mathbf{w}_1 = g(\mathbf{r}_1) \odot 1^{-1} = g(\mathbf{r}_1) \ .$$

Thus the correct aggregate signature σ_1 is

$$\sigma_1 = \pi_1^{-1}(\mathbf{w}_1) = \pi_1^{-1}(g(\mathbf{r}_1)) \ .$$

If i^* is greater than 1, $\mathcal{B}$ calculates the hash $\mathbf{w}_{i^*}$ as

$$\mathbf{w}_{i^*} = g(\mathbf{r}_{i^*}) \odot (\mathbf{r}_{i^*-1})^{-1} \ ,$$

and thus

$$\sigma_{i^*} = \pi_{i^*}^{-1}(\mathbf{w}_{i^*} \odot \sigma_{i^*-1}) = \pi_{i^*}^{-1}(\mathbf{w}_{i^*} \odot \mathbf{r}_{i^*-1}) = \pi_{i^*}^{-1}(g(\mathbf{r}_{i^*})) \ ,$$

where the first substitution follows from the first half of this claim. Thus the claim also holds for $j = i^* > 1$. □

Using the claims above, we can demonstrate that $\mathcal{B}$ correctly answers $\mathcal{A}$'s aggregate signing queries, and that, except when it declares failure, $\mathcal{B}$ correctly computes a claw $\pi(x) = g(y)$, the solution to the challenge posed it.

Claim 4. *If $\mathcal{A}$ makes a valid sequential aggregate query, supplying messages $\mathbf{M}$, keys $\mathbf{s}$, and sequential aggregate signature σ' on all but the last message, then $\mathcal{B}$ either declares failure and halts or outputs the correct sequential aggregate signature σ on the messages.*

Proof. If the request is valid then no key appears twice in $\mathbf{s}$, $|\mathbf{s}| = |\mathbf{M}| = i \leq n$, and $\mathbf{s}_i = s$. Algorithm $\mathcal{B}$ examines $\mathbf{c}_i$. If $\mathbf{c}_i$ equals 0, $\mathcal{B}$ declares failure and exits; if it equals 1, $\mathcal{B}$ outputs $\mathbf{r}_i$ as the answer to the signature query. In this case, the antecedent of Claim 2 is satisfied, and $\sigma = \sigma_i$ equals $\mathbf{r}_i$, as required. □

Claim 5. *If $\mathcal{A}$ outputs a valid and nontrivial aggregate signature forgery σ on messages $\mathbf{M}$ under keys $\mathbf{s}$ then $\mathcal{B}$ either declares failure and halts, or outputs the correct solution x to the given challenge.*

Proof. If the forgery is valid and nontrivial, then no key appears twice in $\mathbf{s}$, $|\mathbf{s}| = |\mathbf{M}| = i \leq n$, and $\mathbf{s}_{i^*} = s$ for some i^*. Algorithm $\mathcal{B}$ examines $\mathbf{c}_{i^*}$. If $\mathbf{c}_{i^*}$ equals 1, $\mathcal{B}$ declares failure and exits. If $\mathbf{c}_{i^*}$ equals 0, the antecedent of Claim 3 is satisfied, and

$$\sigma_{i^*} = \pi_{i^*}^{-1}(g(\mathbf{r}_{i^*})) \ .$$

That is,

$$\pi(\sigma_{i^*}) = g(\mathbf{r}_{i^*}) \ ,$$

where we note that $\pi_{i^*}(\cdot) = \pi(\cdot)$, the challenge permutation. Algorithm $\mathcal{B}$ outputs (in our notation) $x = \sigma_{i^*}$ and $y = \mathbf{r}_{i^*}$; it therefore outputs a claw on $\pi(\cdot)$ and $g(\cdot)$, as required. □

It remains to show that $\mathcal{B}$ outputs the claw with probability at least ϵ'. To do so, we analyze the three events needed for $\mathcal{B}$ to succeed:

$\mathcal{E}_1$: $\mathcal{B}$ does not abort as a result of any of $\mathcal{A}$'s sequential aggregate signature queries.

$\mathcal{E}_2$: $\mathcal{A}$ generates a valid and nontrivial sequential aggregate forgery σ on messages $\mathbf{M}$ under keys $\mathbf{s}$.

$\mathcal{E}_3$: Event $\mathcal{E}_2$ holds, and $c = 0$ for the tuple containing $(\mathbf{s}|_1^{i^*}, \mathbf{M}|_1^{i^*})$ on the H-list, where i^* is the index of s in $\mathbf{s}$.

$\mathcal{B}$ succeeds if all of these events happen. The probability $\Pr[\mathcal{E}_1 \wedge \mathcal{E}_3]$ decomposes as

$$\Pr[\mathcal{E}_1 \wedge \mathcal{E}_3] = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \cdot \Pr[\mathcal{E}_3 \mid \mathcal{E}_1 \wedge \mathcal{E}_2] \ . \tag{1}$$

The following claims give a lower bound for each of these terms.

Claim 6. *The probability that algorithm $\mathcal{B}$ does not abort as a result of $\mathcal{A}$'s aggregate signature queries is at least $1/e$. Hence, $\Pr[\mathcal{E}_1] \geq 1/e$.*

Proof. Without loss of generality we assume that $\mathcal{A}$ does not ask for the signature of the same message twice. We prove by induction that after $\mathcal{A}$ makes k signature queries the probability that $\mathcal{B}$ does not abort is at least $(1 - 1/(q_S + 1))^k$. The claim is trivially true for $k = 0$. Let $(\mathbf{s}^{(k)}, \mathbf{M}^{(k)}$ be $\mathcal{A}$'s k'th signature query and let $\langle \mathbf{s}^{(k)}, \mathbf{M}^{(k)}, w^{(k)}, r^{(k)}, c^{(k)} \rangle$ be the corresponding tuple on the H-list. Then, prior to $\mathcal{A}$'s issuing the query, the bit $c^{(k)}$ is independent of $\mathcal{A}$'s view — the only value that could be given to $\mathcal{A}$ that depends on $c^{(k)}$ is $H(\mathbf{s}^{(k)}, \mathbf{M}^{(k)})$, but the distribution of $H(\mathbf{s}^{(k)}, \mathbf{M}^{(k)})$ is the same whether $c^{(k)} = 0$ or $c^{(k)} = 1$. Therefore, the probability that this query causes $\mathcal{B}$ to abort is at most $1/(q_S + 1)$, the probability that $c^{(k)}$ equals 0. Using the inductive hypothesis and the independence of $c^{(k)}$, the probability that $\mathcal{B}$ does not abort after this query is at least $(1 - 1/(q_S + 1))^k$. This proves the inductive claim. Since $\mathcal{A}$ makes at most q_S signature queries, the probability that $\mathcal{B}$ does not abort as a result of all signature queries is at least $(1 - 1/(q_S + 1))^{q_S} \geq 1/e$. Hence $\Pr[\mathcal{E}_1] \geq 1/e$. □

Claim 7. *If algorithm $\mathcal{B}$ does not abort as a result of $\mathcal{A}$'s queries then algorithm $\mathcal{A}$'s view is identical to its view in the real attack. Hence, $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq \epsilon$.*

Proof. The public key given to $\mathcal{A}$ is from the same distribution as public keys produced by algorithm *KeyGen*. Responses to hash queries are as in the real attack since each response is uniformly and independently distributed in D. All responses to sequential aggregate signature queries are valid. Therefore $\mathcal{A}$ will produce a valid and nontrivial aggregate signature forgery with probability at least ϵ. Hence $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq \epsilon$. □

Claim 8. *The probability that algorithm $\mathcal{B}$ does not abort after $\mathcal{A}$, outputs a valid and nontrivial forgery is at least $1/(q_S + 1)$. Hence, $\Pr[\mathcal{E}_3 \mid \mathcal{E}_1 \wedge \mathcal{E}_2] \geq 1/(q_S + 1)$.*

Proof. Given that events $\mathcal{E}_1$ and $\mathcal{E}_2$ occurred, $\mathcal{B}$ will abort only if $\mathcal{A}$ generates a forgery $(\mathbf{s}, \mathbf{M}, \sigma)$ for which the tuple $\left\langle \mathbf{s}|_1^{i^*}, \mathbf{M}|_1^{i^*}, w^{(i^*)}, r^{(i^*)}, c^{(i^*)} \right\rangle$ on the H-list has $c^{(i^*)} = 1$, where i^* is the index of s in $\mathbf{s}$. At the time $\mathcal{A}$ generates its output, it knows the value of c for those vector pairs $(\mathbf{s}', \mathbf{M}')$ on which it issued a sequential aggregate signature query (and in which s is necessarily the last key). All the remaining c's are independent of $\mathcal{A}$'s view. Indeed, if $\mathcal{A}$ did not issue a signature query for $(\mathbf{s}|_1^{i^*}, \mathbf{M}|_1^{i^*})$, then the only value given to $\mathcal{A}$ that depends on $c^{(i^*)}$ is $H(\mathbf{s}|_1^{i^*}, \mathbf{M}|_1^{i^*})$, but the distribution on $H(\mathbf{s}|_1^{i^*}, \mathbf{M}|_1^{i^*})$ is the same whether $c^{(i^*)} = 0$ or $c^{(i^*)} = 1$. Since the forgery is nontrivial, $\mathcal{A}$ could not have issued a signature query at $(\mathbf{s}|_1^{i^*}, \mathbf{M}|_1^{i^*})$, so $c^{(i^*)}$ is independent of $\mathcal{A}$'s current view and therefore $\Pr[c = 0 \mid \mathcal{E}_1 \wedge \mathcal{E}_2] \geq 1/(q_S + 1)$ as required. □

Using the bounds from the claims above in equation (1) shows that $\mathcal{B}$ produces the correct answer with probability at least $1/e \cdot \epsilon \cdot 1/(q_S + 1)$, as required.

Algorithm $\mathcal{B}$'s running time is the same as $\mathcal{A}$'s running time plus the time is takes to respond to up to q_H hash queries and q_S aggregate signature queries. Each hash query may require as many as n levels of recursion, and each level requires (at most) choosing a random value from D or E, a call to *Evaluate* or *EvaluateG*, an inversion in D, and a evaluation of the group operation $\odot$ in D. Any of these operations is computable in unit time, so each hash query requires at most $4n$ time units to answer. Each signature query involves a corresponding hash computation, and so requires at most $4n$ time units to answer (σ' can be verified at no cost by comparing it to $\mathbf{r}_{i-1}$). Transforming a forgery into a claw (x, y) requires a hash query and a signature verification. As before, the hash query takes at most $4n$ time units to process. The signature verification requires at most n steps, each of which requires a call to *Evaluate*, an inversion in D, and a evaluation of the group operation $\odot$ in D, and thus takes at most n time units to process. The output step thus takes at most $7n$ time units in total. Hence $\mathcal{B}$'s total running time is at most $t + (4nq_H + 4nq_S + 7n) \leq t'$ as required.

In the case when case $\mathcal{B}$ is modified for homomorphic trapdoor permutations, the running-time accounting requires some care, since it needs now two time units to compute *EvaluateG*, not one. Answering a hash oracle query $(\mathbf{s}, \mathbf{M})$ may involve up to n nested computations, but only one entry in $\mathbf{s}$ can contain the challenge key s and require a call to *EvaluateG*. The same is true of the hashing required to answer signature oracle queries and in the output phase of $\mathcal{B}$. In addition, $\mathcal{B}'$ takes 2 time units to compute $\pi^{-1}(z)$. Hence the total running time of $\mathcal{B}'$ is at most $t + ((4n+1)q_H + (4n+1)q_S + 7n + 3) \leq t'$ as required.

Finally, when $\mathcal{B}$ is modified for plain trapdoor permutations, we analyze the running time and the success probability as follows. The challenge z is embedded in only one hash response $(\mathbf{s}, \mathbf{M})$. If $\mathcal{A}$ asks for a signature on $(\mathbf{s}, \mathbf{M})$, it cannot later forge on it — the forgery would be trivial — and so $\mathcal{B}$ can then never succeed in inverting z, and its not being able to answer $\mathcal{A}$'s query is of no consequence. Algorithm $\mathcal{B}''$ succeeds if $\mathcal{A}$ succeeds in creating a forgery, which happens with probability ϵ, and if that forgery includes the challenge $(\mathbf{s}, \mathbf{M})$, which happens with probability at least $1/(q_H + q_S + 1)$. These two probabilities are independent since the placement of the challenge is independent of $\mathcal{A}$'s view. The running time of $\mathcal{B}$ does not change. (The only difference is that, for the single hash query for which $c = 0$, $\mathcal{B}$ need not compute *EvaluateG*, saving one time unit overall).