Sequential Aggregate Signatures with Lazy Verification

Kyle Brogle Sharon Goldberg Leonid Reyzin
Boston University

May 7, 2011

Abstract

Sequential aggregate signature schemes allow n signers, in order, to sign a message each, at a lower total cost than the cost of n individual signatures. We present a sequential aggregate signature scheme based on trapdoor permutations (*e.g.,* RSA). Unlike prior such proposals, our scheme does not require a signer to retrieve the keys of other signers and verify the aggregate-so-far before adding its own signature. Indeed, we do not even require a signer to *know* the public keys of other signers!

Moreover, for applications that require signers to verify the aggregate anyway, our schemes support *lazy verification*: a signer can add its own signature to an unverified aggregate and forward it along immediately, postponing verification until load permits or the necessary public keys are obtained. This is especially important for applications where signers must maintain a large, secure, and current cache of public keys in order to verify messages.

We report a technical analysis of our scheme (which is provably secure in the random oracle model), a detailed implementation-level specification, and implementation results based on RSA and OpenSSL. To evaluate the performance of our scheme, we focus on the target application of BGPsec (formerly known as Secure BGP), a protocol designed for securing the global Internet routing system. There is a particular need for lazy verification with BGPsec, since it is run on routers that must process signatures extremely quickly, while maintaining a large cache of over $36,000$ public keys. We compare our scheme to the algorithms currently proposed for use in BGPsec, and find that our signatures are considerably shorter nonaggregate RSA (with the same sign and verify times) and have an order of magnitude faster verification than nonaggregate ECDSA, although ECDSA has shorter signatures when the number of signers is small.

Contents

1 Introduction

Aggregate signatures schemes allow n signers to produce a digital signature that authenticates n messages, one from each signer. This can be securely accomplished by simply concatenating together n ordinary digital signatures, individually produced by each signer. An aggregate signature is designed to maintain the *security* of this basic approach, while having length much shorter than n individual signatures. To achieve this, many prior schemes *e.g.,* [LMRS04, Nev08] relied on a seemingly innocuous assumption; namely, that each signer needs to *verify* the aggregate signature so far, before adding its own signature on a new message. In this paper, we argue that this can make existing schemes inviable for many practical applications, (in particular, for BGPsec [Lep11] / Secure BGP [KLS00]) and present a new scheme based on trapdoor permutations like RSA that avoids this assumption. In fact, our scheme remains secure even if a signer does not *know* the public keys of the other signers.

1.1 Prior work on aggregate signatures

Boneh, Gentry, Lynn, and Shacham [BGLS03] introduced the notion of aggregate signatures, in which individual signatures could be combined by *any third party* into a single constant-length aggregate. The [BGLS03] scheme is based on the bilinear Diffie-Hellman assumption in the random oracle model[1] [BR93]. Subsequent schemes [LMRS04, Nev08] were designed for the more standard assumption of trapdoor permutations (*e.g.,* as RSA [RSA78]), but in a more restricted framework where third-party aggregation is not possible. Instead, the signers work *sequentially*; each signer receives the aggregate-so-far from the previous signer and adds its own signature.

Lysyanskaya, Micali, Reyzin, and Shacham [LMRS04] constructed the first sequential aggregate signature scheme from trapdoor permutation, with a proof in the random oracle model.[2] However, their scheme has two drawbacks: the trapdoor permutation must be *certified* (when instantiating the trapdoor permutation with RSA, this means that each signer must either prove certain properties of the secret key or else use a long RSA verification exponent), and each signer needs to verify the aggregate-so-far before adding its own signature. Neven [Nev08] improved on [LMRS04] by removing the need for certified trapdoor permutations, but the need to verify before signing remained. Indeed, a signer who adds its own signature to an unverified aggregate in both [LMRS04] and [Nev08] is exposed to a devastating attack: an adversary can issue a single malformed aggregate to the signer, and use the signature on that malformed message to generate a *valid* signature on a message that the signer never intended to sign (Appendix A).

Thus, the advantages of basing the schemes on trapdoor permutations (particularly a more standard security assumption and fast verification using low-exponent RSA) are offset by the disadvantage of requiring verification before signing. We argue below that this disadvantage is serious.

1.2 The need for lazy verification

In applications with a large number of possible signers, the need to verify before signing can introduce a significant computational bottleneck; each signer must quickly retrieve the public keys of the signers involved in the aggregate-so-far before it can even begin to run its signing algorithm.

[1]The need for the random oracle model was removed by Lu, Ostrovsky, Sahai, Shacham, and Waters [LOS^{+}06], who also relied on the bilinear Diffie-Hellman assumption; however, this improvement in security came at a considerable efficiency cost—see [CHKM10] for a detailed analysis. See also [RS09, CSC09] for other proposals based on less common assumptions.

[2]Bellare, Namprempre, and Neven [BNN07] showed how the schemes of [BGLS03] and [LMRS04] can be improved through better proofs and slight modifications.

Worse yet, signers need to keep their large caches of public keys secure and current: if a public key is revoked and a new one is issued, the signer must first obtain the new key and verify its certificate before signing the aggregate-so-far.

A key application: BGPsec. Sequential aggregate signatures are particularly well-suited for the BGPsec [Lep11] (formerly known as the Secure Border Gateway Protocol (S-BGP) [KLS00]), a protocol designed to improve the security of the global Internet routing system. (This application was mentioned in [BGLS03, Nev08] among others, and explored further in [ZSN05]; we should note that identity-based aggregate signatures are also being proposed for this application—see, e.g., [BJ10, SVJR10] and references therein.) In BGPsec, autonomous systems (ASes) digitally sign routing announcements listing the ASes on the path to a particular destination. An announcement for a path that is n hops long will contain n digital signatures, added *in sequence* by each AS on the path. BGPsec naturally requires routers to store and maintain a large local cache of public keys; indeed, a routing announcement can contain information from *any* of the 36,000 ASes in the Internet [COZ08]. Given the difficulty of storing, retrieving, and verifying certificates for $> 36K$ public keys, the BGPsec protocol gives routers the option to perform *lazy verification*: that is, to immediately sign the routing announcement with its *own* public key, and to delay verification until a later time, *e.g.,* when (a) it has time to retrieve the public keys of the other signers, or (b) when the router itself is less overloaded and can devote resources to verification [DHS]. Requiring that routers delay signing and re-announcing BGPsec messages until verification is complete is a non-starter, as it adds latency, introducing problems with router performance and global protocol convergence. Indeed, lazy verification has been written into the BGPsec specification [Lep11]:

> ...it is important to note that when a BGPSEC speaker signs an outgoing update message, it is not attesting to a belief that all signatures prior to its are valid.

There is legitimate concern that lazy verification might cause routers to temporarily adopt unverified paths. However, without lazy verification, incremental deployment of BGPSec becomes infeasible, particularly because it must be run on legacy routers. Thus, any signature scheme adopted by BGPsec must fulfill this requirement.

No public keys in the signing algorithm! Note that the primary obstacle here is *not* verification time (which, with low-exponent RSA, can be considerably faster than signing time), but the need to obtain public keys. Thus, lazy verification also requires that prior signers' public keys are *not* used in the signing algorithm (*e.g.,* hashed with the message as in [LMRS04, Nev08]).

1.3 Overview of our contributions

We present a sequential aggregate signature scheme with lazy verification, based on any trapdoor permutation (such as RSA). Moreover, as in the nonsequential scheme of [BGLS03], our signers do not need to know anything about each other—not even each other's public keys. To achieve this, we modify Neven's scheme [Nev08] by randomizing the H-hash function with a fresh random string per signer, which becomes a part of the signature, similarly to Coron's PFDH [Cor02] (Section 3). Thus, unlike existing schemes that have *constant-length* aggregates, our aggregate grows *linearly* with the number of signers; however, as we discuss below, this growth in length is small. Further, we show that the length of the per-signer random string can be reduced if the randomness is input-dependent (Section 5). This modification allows the i^{th} signer to sign without verifying, and without even needing to know the public keys of all the signers that came before him. We make the following contributions:

Generic randomized scheme. We present the basic version of our scheme, which requires each signer to append a *truly random* string to the aggregate (Section 3). Our scheme is as efficient for signing and verifying (per signer) as ordinary trapdoor-permutation based signatures, like the Full-Domain-Hash (FDH, [BR93, Section4]). We prove security (Section 4) in the random oracle model, based on the same assumption of trapdoor permutations (or claw-free permutations for a tighter security reduction) as in [Nev08]. Our security proof is more involved, because the reduction cannot know the public keys of other (adversarial) signers during the signature queries. We should note that our proof technique also shows that Neven's original scheme need not hash other signer's public keys in the signing algorithm (however, Neven's original scheme still fails under lazy verification).

Generic input-dependent randomness scheme. Next, we use a combinatorial argument to show the signature can be shortened if the random strings are computed as a function of the input (Section 5).

Instantiating with RSA. Appendix G shows how to instantiate our schemes with practical trapdoor permutations like RSA, which have slightly different domains for different signers.

Detailed specification. We provide a full, parameterized step-by-step specification of the truly-random and input-dependent-random versions of our signature when instantiated with RSA (Appendix H). We also provide guidelines on choosing parameters such as bit lengths (Section 6.1).

Implementation, benchmarking and practical considerations. We implement our specification as a module in OpenSSL (Section 6); the implementation is available from [BGR11]. We compare our implementation's performance to other potential solutions that allow for lazy verification; namely, [BGLS03], and the "trivial" solution of using n RSA or ECDSA signatures (the two algorithms currently proposed for use in implementations of BGPsec [DHS]). When evaluating signatures schemes for use with BGPsec, we consider compute time as well as signature length. Thus, we show that our signature is shorter than trivial RSA when there are $n > 1$ signers and shorter than trivial ECDSA when there are $n > 6$ signers. (While our signature is longer than the constant-length [BGLS03] signature, it benefits from relying on the better-understood security assumption of RSA.) Moreover, our scheme enjoys the same extremely fast verify times as RSA, which is crucial for applications like BGPsec, where the i^{th} signer may need to verify $i - 1$ times for each time it signs.

2 Preliminaries

Sequential aggregate signature security. The security definition for aggregate signatures is designed to capture the following intuition: each signer is individually secure against existential forgery following an adaptive chosen-message attack [GMR88] regardless of what all the other signers do. In fact, we will allow the adversary to give the attacked signer arbitrary—perhaps meaningless—aggregate-so-far signatures during the signature queries, thus making them adaptive "chosen-message-and-aggregate" queries. We also allow the adversary, which we often refer to as "the forger" to choose the public keys of all the other signers and to place the single signer who is under attack anywhere in the signature chain in the attempted forgery.

Our definition, given in Appendix B, is almost verbatim from [LMRS04], with three important differences: (i) the public keys of previous signers are not input to the signing algorithm; (ii) the forger, in its query to ith signer, is required to supply only the aggregate-so-far, but not the messages or public keys with respect to which this aggregate was allegedly produced; and (iii) to be considered successful, the forger must forge a signature on a new message against an uncorrupted signer—in other words, it is not enough to change a public key or message of someone else in the chain before the corrupted signer (because such public keys and messages are not even specified during the signing query).

Cryptographic primitives. We will use pseudorandom function [GGM86] defined for the sake of completeness in Appendix C. We denote by $\varepsilon_{\mathsf{PRF}}(q,t)$ the maximum advantage that an adversary who asks q queries and runs in time t has in distinguishing the pseudorandom family PRF from a truly random function. We assume the reader is familiar with the trapdoor and claw-free permutations; we will denote by π the easy direction of the trapdoor permutation, by π^{-1} the hard direction, and by ρ the function such that it is hard to find a "claw" x, z with $\pi(x) = \rho(z)$.

3 Our basic signature scheme

We present the basic version of our signature scheme. Because sending messages in the clear is necessary for lazy verification (*i.e.,* to use the message before its authenticity is confirmed), we do not consider signatures with message recovery [Nev08]. However, a version with message recovery, which saves some space, can be constructed similarly to [Nev08]. We use the following notation:

- Let m_i be the message signed by signer i.
- Let trapdoor permutation π_i be the public key of signer i and π_i^{-1} be the corresponding secret key. We assume all permutations operate on bit strings of length ℓ_π, *i.e.,* have domain and range $\{0,1\}^{\ell_\pi}$. (In Appendix G we remove the assumption that all permutations operate on the same domain. Section 6 uses this to instantiate π from the RSA assumption, where π_i is the easy direction, and π_i^{-1} is the hard direction of the RSA permutation.)
- Let H (*resp.* G) be a cryptographic hash function (modeled as a random oracle) that outputs ℓ_H-bit (*resp.* ℓ_π-bit) strings.
- Let ℓ_r be a parameter denoting the length of the randomness appended by each signer.
- Let the notation $\vec{a_i}$ denote a vector of values $(a_1, a_2, ..., a_i)$.
- Let $\oplus$ to denote bitwise exclusive-or. Exclusive-or is not the only operation that can be used; any efficiently computable group operation with efficient inverse can be used here.
- ϵ is a special character denoting the empty string; we assume $\epsilon \oplus x = x$ for any x.

Algorithm 1 Sign: The i^{th} Signer's algorithm

```
Require: $\pi_i, m_i, x_{i-1}, h_{i-1}$ (where $x_{i-1}, h_{i-1} = \epsilon, \epsilon$ if $i = 1$).
1: Draw $r_i \overset{R}{\leftarrow} \{0,1\}^{\ell_r}$
2: $\eta_i \leftarrow H(\pi_i, m_i, r_i, x_{i-1})$
3: $h_i \leftarrow h_{i-1} \oplus \eta_i$
4: $g_i \leftarrow G(h_i)$
5: $y_i = g_i \oplus x_{i-1}$
6: $x_i \leftarrow \pi_i^{-1}(y_i)$
7: return $r_i, x_i, h_i$
```

The i^{th} signer's signing algorithm (Algorithm 1) is *strictly* constant in the number of signers; it takes in *only* the i^{th} signers' own public key and message and the aggregated portion of the signature x_{i-1}, h_{i-1}. Moreover, the aggregated signature need not be verified before it is signed. For verification (Algorithm 2), only a single x_i and h_i—namely, the one from the last signer—is needed. However, every r_i, from the first signer to the last, is needed.

Algorithm 2 $\mathsf{Ver}^{H,G}$: The Verification Algorithm

```
Require: $\vec{\pi_n}, \vec{m_n}, \vec{r_n}, x_n, h_n$
1: for $i = n, n-1, ...., 2$ do
2:     $y_i \leftarrow \pi_i(x_i)$
3:     $g_i \leftarrow G(h_i)$
4:     $x_{i-1} \leftarrow g_i \oplus y_i$
5:     $\eta_i \leftarrow H(\pi_i, m_i, r_i, x_{i-1})$
6:     $h_{i-1} \leftarrow h_i \oplus \eta_i$
7: if $h_1 = H(\pi_1, r_1, m_1, \epsilon)$ and $\pi_1(x_1) = G(h_1)$ then
8:     return 1
9: else
10:    return 0
```

4 Security proof

We prove security for the claw-free permutation case (the more general trapdoor permutation case results in a looser security reduction, with a multiplicative loss of q_H; since we propose to instantiate our scheme with RSA, which is claw-free, we do not present the more general case here, but the differences in the proof are straightforward; see, e.g., [DR02]). Our proof shows how a forger F on the aggregate signature scheme can be used to construct a reduction R that finds a claw in claw-free pair (π_*, ρ_*). R has F forge a signature for victim signer that uses permutation π_*, and then uses the resulting forgery to find the claw in the claw-free pair. The structure of our reduction is similar to [Nev08]; however, while [Nev08] constructs a "sequential forger" from forger F and then constructs reduction R from the sequential forger, our reduction must proceed in one step (since the notion of a sequential forger is undefined if hash queries do not include previous signers public keys).

F's queries. We review what forger F expects to see on each one of its queries:

- **H-Query.** F asks query $Q = (\pi, m, r, x)$ (where x may be ϵ) and expects to see $H(Q) = \eta$.
- **G-query.** F asks query h, and expects to see $g = G(h)$.
- **Sign Query.** F asks query (m, h, x) to be signed by π_*, and expects to see h', x' back, where $h' = h \oplus H(\pi_*, m, r, x)$ for some (random-looking) r and $\pi_*(x') = G(h') \oplus x$.
- **Forgery.** Finally, F outputs a forgery, $\sigma = \vec{\pi_n}, \vec{m_n}, \vec{r_n}, x_n, h_n$ where $\pi_n = \pi_*$. (Value n is chosen by F).

Simplifying assumptions about the forger F. The following simplifies our proof:

- We assume that the forger F forges the last signature in the signature chain; in other words, $\pi_n = \pi_*$ and m_n is a new message never queried by F to the signing oracle (whose public key is π_*). Indeed, any F can be easily modified to do so: if π_* and a new message $m_{n'}$ are present in $\vec{\pi_n}$ but at location $n' < n$, then we can run the verification algorithm loop for $n - n'$ iterations to obtain $x_{n'}, h_{n'}$ and output $\sigma' = \vec{\pi_{n'}}, \vec{m_{n'}}, \vec{r_{n'}}, x_{n'}, h_{n'}$ as the new forgery, which will be valid if an only if σ was valid. Note that we do not assume that π_* (or any other public key) is present in the signature chain only once.
- We assume that before forger F outputs its forgery and halts, it makes hash queries on all the hashes that will be computed during the verification of its forgery. Moreover, we assume that the forger does not output an invalid forgery; instead, it halts and outputs $\perp$. Indeed,

any F can be modified to do so; simply run the verification algorithm upon producing the forgery, and check that m_n is different from every message asked in a sign query.

4.1 Description of the reduction R

4.1.1 Data structures used by R

HT and GT tables. The reduction R uses 'programmable random oracles', *i.e.,* it chooses answers for random oracle queries. R keeps track of queries whose answers have already been decided in two tables: HT for H and GT for G. We say $\mathsf{HT}(Q) = \eta$ if HT stores η as the answer to a query Q, and $\mathsf{HT}(Q) = \perp$ if HT has no answer for Q (similar for GT).

The HTree. The key challenge for the reduction is programming G, since G-queries are made on sums of H-query answers, rather than on individual H-query answers. Thus the reduction keeps an additional data structure, the HTree, that records responses to H-queries that may eventually be used as part of forger F's forgery. (HTree is inspired by the graph $\mathcal{G}$ in [Nev08, Lemma 5.3].)

The HTree is a tree of labeled nodes that stores a subset of the queries in HT. Each node in HTree (except the root) corresponds to an H-query that could potentially appear in the forger F's final forgery σ; the queries asked during verification of σ will appear on a path from one of the leaf nodes to the root (unless a very unlikely event occurs). The HTree has a designated *root node* that stores the value $h_0 = 0$. We consider the root to be at depth 0. A node N_i at depth $i > 0$ stores:

- a pointer to its parent node
- a query $Q_i = (\pi_i, m_i, r_i, x_{i-1})$ (where $x_{i-1} = \epsilon$ if and only if $i = 1$),
- the 'hash-response' values η_i and h_i (h_i is the XOR of the values $\eta_1, \dots, \eta_i$ on the path from the root to the node N_i; equivalently, $h_{i-1} \oplus \eta_i$, where h_{i-1} is stored in the parent node),
- an auxiliary value y_i that is used to determine how future queries are added to the HTree, computed as $G(h_i) \oplus x_{i-1}$ (note that y_i is the value to which the signer would apply π_i^{-1}),
- if $\pi_i = \pi_*$, an auxiliary value z that may be used to find a claw in (π_*, ρ_*).

Every node at depth $i = 2$ or deeper satisfies the relation $\pi_{i-1}(x_{i-1}) = y_{i-1}$, where π_{i-1} and y_{i-1} are stored at the node's parent. New H-queries Q are added as nodes to the HTree if they can satisfy this relation; we say that such a query can be *tethered* to an existing node in the HTree. Intuitively, a query tethered to N_i becomes a child of N_i in the HTree:

Definition 4.1 (Tethered queries)**.** *An H-query Q containing $x \neq \epsilon$ is* tethered *to node N_i in the HTree if N_i stores π_i, y_i such that $\pi_i(x) = y_i$. If $x = \epsilon$, then Q is tethered to the root of the HTree.*

The HTree's Lookup function (Algorithm 3) determines the HTree node to which query Q can be tethered. (Lemma D.3 argues that Lookup finds at most *one node* with high probability.) The HTree is populated via the $\mathsf{Sim\text{-}H}$ algorithm (Algorithm 6). The reduction R adds an H-query Q to the HTree if and only if it is tethered to some node in the HTree *at the time that forger F makes the H-query.* It is possible that some query Q is not tethered at the time it is made, but becomes tethered at at *later* time (after some new nodes are added to the HTree). However, Claim 4.4 shows that this is highly unlikely.

4.1.2 Algorithms used to answer forger's queries

The reduction R uses the following algorithms (Algorithms 3–7) to answer each of forger F's queries.

G-queries. R answers these queries using a simple algorithm $\mathsf{Sim\text{-}G}$ (Algorithm 4). $\mathsf{Sim\text{-}G}$ returns $\mathsf{GT}(h)$ if it is already defined, or, if not, returns a fresh random value and records it in the GT.

Sign-queries The reduction R answers queries (m, h, x) to be signed by π_* using $\mathsf{Sim\text{-}S}$ (Algorithm 7). Since the reduction does not know the inverse of the challenge permutation π_*^{-1}, it 'fakes' a valid signature by carefully assigning certain entries in random oracle tables HT, GT, and ABORTS if these entries in HT, GT have been previously assigned. Later, we argue that $\mathsf{Sim\text{-}S}$ is unlikely to abort, since the entries added to HT, GT by $\mathsf{Sim\text{-}S}$ depend on a fresh random value r chosen as part of each signature query (Lemma D.6).

H-queries The reduction R answers these queries $Q = (\pi, m, r, x)$ using the $\mathsf{Sim\text{-}H}$ (Algorithm 6). If there is an entry for Q in the HT, then $\mathsf{Sim\text{-}H}$ returns it. Otherwise, it assigns a fresh random value η as $\mathsf{HT}(Q)$. Next, $\mathsf{Sim\text{-}H}$ needs to prepare for the event that Q could lead to a forgery by the forger F, and thus needs to be stored in the HTree. To do this, $\mathsf{Sim\text{-}H}$ uses the Lookup function to check if Q can be tethered and thus should be added to the HTree. If Q can be tethered, $\mathsf{Sim\text{-}H}$ adds a new node to the HTree containing Q, its hash response η, and an axillary value y that is used by the Lookup function to tether future H-queries. In order to ensure that HTree is a tree (Lemma D.3), it is important to ensure that y is a fresh random value; $\mathsf{Sim\text{-}H}$ aborts if that's not the case. Finally, if Q contains the challenge permutation π_*, $\mathsf{Sim\text{-}H}$ adds a value z to the HTree node that $\mathsf{FindClaw}$ will use to derive a claw from a valid forgery output by the forger F. To prepare these values, $\mathsf{Sim\text{-}H}$ behaves almost as if it is 'faking' the answer to a sign-query, except that instead of using the usual challenge permutation π_* (as in $\mathsf{Sim\text{-}S}$), it uses the challenge permutation ρ_* applied to z (so as to benefit from forger F's forgery, which would invert π_* on the output of $\rho_*(z)$, thus producing a claw). As in $\mathsf{Sim\text{-}S}$, this involves carefully assigning certain entries in GT, and aborting if these entries are already assigned. (Claim D.5 shows that $\mathsf{Sim\text{-}H}$ is unlikely to abort.)

Finding a claw. Finally, forger F outputs a forgery $\vec{\pi_n}, \vec{m_n}, \vec{r_n}, x_n, h_n$, where $\pi_n = \pi_*$. Recall that our simplifying assumptions mean that the forgery is valid. The reduction R uses $\mathsf{FindClaw}$ (Algorithm 5) to find a claw from the forgery. Because we assumed all the queries for verifying σ have already been asked, the query $(pi_*, m_n, r_n, x_{n-1})$ is in HT. Moreover, if the forgery is valid, then with high probability it is in the HTree as a child of the node storing $(\pi_{n-1}, m_{n-1}, r_{n-1}, x_{n-2})$, which is in turn a child of the node storing $(\pi_{n-2}, m_{n-2}, r_{n-2}, x_{n-3})$, *etc.*. This holds because in a valid forgery, each H-query made during verification is tethered to the next one, and, by Claim 4.4, all tethered queries are in the HTree with high probability. The value x_n (from the forgery σ) and value z_n (from HTree node of the query $Q = (\pi_*, m_n, r_n, x_{n-1})$) constitute a claw.

Algorithm 3 Lookup

```
Require: $x$
 1: if $x = \epsilon$ then
 2:     return Root node of HTree
 3: else
 4:     Nodelist = {all nodes $N$ in HTree containing $\pi, y$ such that $\pi(x) = y$}
 5:     if Nodelist contains more than one node then
 6:         ABORT
 7:     else if Nodelist is empty then
 8:         return $\perp$
 9:     else
10:         return the single node in Nodelist
```

Algorithm 4 Sim-G: Answering a G-Query

```
Require: $h$
 1: if $\mathsf{GT}(h) = \perp$ then
 2:     Draw $g \overset{R}{\leftarrow} \{0,1\}^{\ell_\pi}$
 3:     $\mathsf{GT}(h) \leftarrow g$
 4: return $\mathsf{GT}(h)$
```

Algorithm 5 FindClaw

```
Require: $\sigma = (\vec{\pi_n}, \vec{m_n}, \vec{r_n}, x_n, h_n)$ with $\pi_n = \pi_*$
 1: $N_n \leftarrow \mathsf{Lookup}(x_n)$
 2: if $N_n = \perp$ then
 3:     ABORT
 4: Retrieve $z_n$ the node $N_n$
 5: return Claw $(x_n, z_n)$.
```

Algorithm 6 Sim-H: Answering an H-Query

```
Require: $Q = (\pi, m, r, x)$
 1: if $\mathsf{HT}(Q) = \perp$ then
 2:     Draw $\eta \overset{R}{\leftarrow} \{0,1\}^{\ell_H}$
 3:     $\mathsf{HT}(Q) \leftarrow \eta$
 4:     $N_{i-1} \leftarrow \mathsf{Lookup}(x)$
 5:     if $N_{i-1} \neq \perp$ then
 6:         Create new node $N_i$ with parent $N_{i-1}$
 7:         Retrieve $h_{i-1}$ from parent $N_{i-1}$
 8:         $h_i \leftarrow h_{i-1} \oplus \eta$
 9:         if $\mathsf{GT}(h_i) \neq \perp$ then
10:             ABORT
11:         if $\pi \neq \pi_*$ then
12:             $g_i \leftarrow \mathsf{Sim\text{-}G}(h_i)$
13:             $y_i \leftarrow g_i \oplus x$
14:             Populate node $N_i$ with $Q, \eta, h_i, y_i$
15:         else
16:             Draw $z_i \overset{R}{\leftarrow} \{0,1\}^{\ell_\pi}$
17:             $y_i \leftarrow \rho_*(z_i)$
18:             Populate node $N_i$ with $Q, \eta, h_i, y_i, z_i$
19:             $\mathsf{GT}(h_i) \leftarrow y_i \oplus x$
20: return $\mathsf{HT}(Q)$
```

Algorithm 7 Sim-S: Answering a Sign-Query

```
Require: $(m, h, x)$
 1: Draw $r \overset{R}{\leftarrow} \{0,1\}^{\ell_r}$
 2: $Q \leftarrow (\pi_*, m, r, x)$
 3: if $\mathsf{HT}(Q) \neq \perp$ then
 4:     ABORT
 5: else
 6:     Draw $\eta \overset{R}{\leftarrow} \{0,1\}^{\ell_H}$
 7:     $\mathsf{HT}(Q) \leftarrow \eta$
 8: $h' \leftarrow \eta \oplus h$
 9: Draw $x' \overset{R}{\leftarrow} \{0,1\}^{\ell_\pi}$
10: $y' \leftarrow \pi_*(x')$ .
11: if $\mathsf{GT}(h') = \perp$ then
12:     $\mathsf{GT}(h') \leftarrow y' \oplus x$
13: else
14:     ABORT
15: return $r, h', x'$.
```

4.2 Analysis of the reduction

Theorem 4.2. *If a forger F is such that $\mathsf{Adv\,AggSig}_F = \varepsilon$, then the reduction R finds a claw for (π_*, ρ_*) in about the same running time as F with probability*

$$\varepsilon - (q_S + q_H)(q_S + q_G + q_H)2^{-\ell_H} - q_S(q_S + q_H)2^{-\ell_r} - q_H^2 2^{-\ell_\pi} \tag{1}$$

where q_H is the number of H-hash queries, q_G is the number of G-hash queries, and q_S is the number of sign queries made by the forger F.

We prove this theorem in Appendix D. The proof hinges on two key statements about the HTree. First, the probability that $\mathsf{Lookup}(x)$ finds more than one HTree node is low. Second, an H-query that was not added to HTree is unlikely to become tethered at *some later time.* Both statements rely on the fact (proven in Claim D.4) that each time a query is placed on the HTree, its y value is random and independent of every other y value. We now present these two claims, that are (arguably) the most interesting parts of the proof:

$\mathsf{Lookup}(x)$ is unlikely to find more than one node. We need to bound the probability that there are nodes N_1 and N_2 in HTree storing π_1, y_1 and π_2, y_2 such that there exists x with $\pi_1(x) = y_1$ and $\pi_2(x) = y_2$. Note that forger F can *adversarially-choose* π_1, π_2 stored in nodes N_1 and N_2 (since F issues a H-query that sets the π_i stored at each HTree node). Indeed, we have the following process: F chooses π_1 first, and then is given a independent random y_1, then chooses π_2 with knowledge of π_1, y_1, and finally is given independent random y_2 (Claim D.4). Thus, we *cannot* assume that π_1, π_2 are *permutations*; however, we may assume that they are *functions*:

Claim 4.3. *For* any *two functions π_1, π_2 with domain $\{0,1\}^{\ell_\pi}$, and two uniformly random values y_1, y_2 in $\{0,1\}^{\ell_\pi}$, there exists x such $\pi_1(x) = y_1$ and $\pi_2(x) = y_2$ with probability at most $2^{-\ell_\pi}$.*

Proof. Define the set of preimages of y_1 under π_1 as $S_{y_1} = \{x \mid \pi_1(x) = (y_1)\}$. Suppose $|S_{y_1}| = \alpha$. Then there are at most α choices of y_2 that will result in the event that there exists x such $\pi_1(x) = y_1$ and $\pi_2(x) = y_2$, because each element $x \in S_{y_1}$ gives rise to at most one $y_2 = \pi_2(x)$. Because y_2 is chosen uniformly from a set of size 2^{ℓ_π}, the probability that x satisfying $\pi_1(x) = y_1$ and $\pi_2(x) = y_2$ exists is at most $\alpha 2^{-\ell_\pi}$. Thus, the desired probability is at most

$$\sum_\alpha \alpha 2^{-\ell_\pi} \Pr_{y_1}[|S_{y_1}| = \alpha] = 2^{-\ell_\pi} \sum_\alpha \alpha \cdot \frac{|\{y_1 \text{ s.t. } |S_{y_1}| = \alpha\}|}{2^{\ell_\pi}}.$$

Observing that $\sum_\alpha \alpha \cdot |\{y_1 \text{ s.t. } |S_{y_1}| = \alpha\}| = |\mathrm{Domain}(\pi_1)| = 2^{\ell_\pi}$, we get the desired bound. $\square$

H-queries are unlikely to get tethered after they are asked. Next, we prove the following:

Claim 4.4. *If an H-query did not get added to HTree (equivalently, if it was untethered at the time it was asked to $\mathsf{Sim\text{-}H}$), then the probability it will ever become tethered is at most $q'_H 2^{-\ell_\pi}$, where q'_H is the number of H queries made after it.*

Proof. Consider queries as they are added to HT in order. Suppose j_0'th query $Q = (\pi, m, r, x)$ was added as the result of a query to $\mathsf{Sim\text{-}H}$ and was untethered at the time it was asked, *i.e.,* the HTree was such that $\mathsf{Lookup}(x) = \perp$. Now suppose that Q first becomes tethered after some j_1-th query, $Q' = (\pi', m', r', x')$, is placed in HT. From the definition of a tethered query, Q' must have been added to the HTree. Thus, we must have $j_1 > j_0$, because we never remove nodes from HTree (*i.e.,* we cannot have $j_1 < j_0$) and Q itself is not added to the HTree (thus, $j_1 \neq j_0$). Since nodes are added to the HTree only when Sim-H is called on a new query, it follows that Q' was added to HTree after forger F asked a new H-query, so that the following collision occurs:

$$\pi'(x) = y' . \tag{2}$$

But, y' is a uniform random value that is independent of π' and x (Claim D.4), so the collision in equation (2) occurs with probability $2^{-|y|} = 2^{-\ell_\pi}$. This holds for each of the q'_H queries that could have been asked after j_0, and the claim follows by the union bound. □

5 Input-dependent randomness

To shorten our signature, we now show how to reduce ℓ_r (the length of the randomness appended by each signer). To do this, we replace the truly random r from our basic scheme with an r that is computed as a function of the inputs to the signer, and argue that it can be made shorter than the random r. Moreover, this input-dependent r need not be truly random; it suffices for a r to be a *psuedorandom* function of the input.

5.1 Modifying the scheme

We now compute r as a pseudorandom function (PRF) over the input (m_i, h_{i-1}, x_{i-1}) received by that signer i. Let $\mathsf{PRF}_{\mathsf{seed}} : \{0,1\}^* \to \{0,1\}^{\ell_r}$ be a pseudorandom function with seed seed and insecurity $\varepsilon_{\mathsf{PRF}}(q,t)$ against adversaries asking q queries and running in time t. Add a uniformly chosen seed to the secret key of the signer and replace line 1 of the signing algorithm (Algorithm 1) with $r \leftarrow \mathsf{PRF}_{\mathsf{seed}}(m, h, x)$.

5.2 Security proof

In the previous section, we found that ℓ_r must be long enough to tolerate a security loss of $q_S(q_H + q_S)2^{-\ell_r}$ (Theorem 4.2). We now show how to reduce ℓ_r so that it need only allow for a security loss of approximately $(q_G + q_H + q_S + \ell_H q_S^2)2^{-\ell_r}$. This is an improvement if we assume that $q_H \approx q_G$ (since both H and G are hash functions) and $q_S \ll q_H$ (since in practice hash queries can be made offline, while signing queries need access to an actual signer):

Theorem 5.1. *If a forger F is such that $\mathsf{Adv\,AggSig}_F = \varepsilon$ for the modified scheme, then the reduction R finds a claw for (π_*, ρ_*) in about the same running time as F with probability*

$$\begin{aligned} &\varepsilon - 2(q_S + q_H)(q_S + q_G + q_H)2^{-\ell_H} - q_H^2 2^{-\ell_\pi} \\ &- (q_G + q_H + q_S + (\ell_h + 2)q_S^2)2^{-\ell_r} - \varepsilon_{\mathit{PRF}}(q_S, t) \end{aligned} \tag{3}$$

where q_H is the number of H-hash queries, q_G is the number of G-hash queries, q_S is the number of sign queries made by the forger F, and t is the running time of the forger and the reduction combined.

The intuition behind this result is as follows. When r was truly random, as in Section 3, we had to choose r long enough to prevent the forger from making a sign query on $(\pi_*, m_i, x_{i-1}, h_{i-1})$ for which Sim-S draws a random r_i that collides with a previously made H-query $Q_i = (\pi_*, m_i, r_i, x_{i-1})$. However, notice that a sign-query also includes the value h_{i-1}! Thus, by computing r_i as a pseudorandom function of h_{i-1}, we make it more difficult for the forger to create a collision that causes Sim-S to abort, because now the collision must include π_*, m_i, r_i, x_{i-1} and h_{i-1}.

Sketch. The full proof of this theorem is in Appendix E. We modify the reduction R to replace the ABORT on line 4 of $\mathsf{Sim\text{-}S}$ (Algorithm 7) with

$$\eta \leftarrow \mathsf{HT}(Q)$$

(All the rest of R's algorithms are unchanged.) As such, we must now consider a new case where $\mathsf{Sim\text{-}S}$ aborts; namely, if $\mathsf{Sim\text{-}S}$ draws an r that defines a query $Q = (\pi_*, m, r, x)$ that is already in HT *and* the value $h' = \eta \oplus h$ is already stored in the GT (where recall that $\eta = \mathsf{HT}(Q)$ and h was given as part of the query to $\mathsf{Sim\text{-}S}$). Call such r *bad* for m, h, x. How likely is it that $\mathsf{Sim\text{-}S}$ draws a bad r?

Claim 5.2. $\Pr[\textit{Sim-S ever draws a bad } r] \leq (q_S + q_H)^2 2^{-\ell_H} + (q_G + q_H + q_S + (\ell_h + 2)q_S^2)2^{-\ell_r}$.

Proof. Consider a matrix ζ whose rows are indexed by queries (*i.e.,* h') in GT and whose columns are indexed by queries in HT that start with π_* (*i.e.,* , there is a column for each $Q = (\pi_*, m, r, x) \in \mathsf{HT}$). The entry in row h' and column Q is $h = h' \oplus \eta$, where $\eta = \mathsf{HT}(Q)$. $\mathsf{Sim\text{-}S}$ draws an r that is bad for m, h, x if and only if it (a) draws an r such that $Q = (\pi_*, m, r, x) \in \mathsf{HT}$, and (b) h exists in the Q^{th} column of ζ.

Thus, we will say that a column $Q = (\pi_*, m, r, x)$ of ζ is *bad* for (m, h, x) if at least one of the entries in that column is h (denote the set of such columns $BAD(m, h, x)$). The number of r values that are bad for a particular triple (m, h, x) is equal to the number of columns that are bad for that triple, and thus the probability that a bad r is chosen by $\mathsf{Sim\text{-}S}$ when responding to (m, h, x) is equal to $2^{-\ell_r} \cdot |BAD(m, h, x)|$. Now consider all the queries that have a given h. Note that the bad columns do not overlap for such queries (because each column is labeled with m and x). By the union bound, the probability that $\mathsf{Sim\text{-}S}$ draws a bad r during any signature query with h is at most $2^{-\ell_r} \cdot \sum_{m,x} |BAD(m, h, x)|$. Since the bad columns do not overlap, $\sum_{m,x} |BAD(m, h, x)|$ is bounded by the number of times h occurs in ζ. Thus, we can bound the overall probability that $\mathsf{Sim\text{-}S}$ ever draws a bad r by at most:

$$2^{-\ell_r} \sum_{i=1}^{q_S} \#\text{ of times the } i^{th} \text{ most frequent entry appears in } \zeta\,.$$

Thus, we have a combinatorial problem to solve:

Combinatorial problem. Suppose β values $\eta_1, \ldots, \eta_\beta$ are chosen uniformly at random as ℓ_H-bit strings and given to an adversary, who then chooses α distinct values $h'_1, \ldots, h'_\alpha$. The $\alpha \times \beta$-matrix ζ is constructed by XORing the η and the h' values. A *collision* in ζ is a set of entries that are all equal. What is the total number of entries in the γ biggest collisions?

Theorem 5.3. *With probability at least* $1 - \beta^2 2^{\ell_H}$*, the total size of the* γ *biggest collisions in* ζ *is at most* $\alpha + (\ell_h + 2)\gamma^2$.

We can use Theorem 5.3 (proved in Appendix F) to bound the probability of choosing a bad r. α is the size of GT, which is at most $q_G + q_H + q_S$. β is the number of HT entries, which is at most $q_S + q_H$. γ is at most q_S. Then, the claim follows by observing that the probability that $\mathsf{Sim\text{-}S}$ ever aborts is at most (a) the probability that the event of Theorem 5.3 doesn't hold, which is at most $\beta^2 2^{-\ell_H} = (q_S + q_H)^2 2^{-\ell_H}$, plus (b) the probability that, even though the event of Theorem 5.3 holds, a bad r is chosen, which is at most $(\alpha + (\ell_h + 2)\gamma^2)2^{-\ell_r} \leq (q_G + q_H + q_S + (\ell_h + 2)q_S^2)2^{-\ell_r}$. □

□

	2048-bit RSA	Our scheme	256-bit ECDSA
Signature length (bits)	$2048n$	$2048 + 256 + (128 + 1)n$	$2 \cdot 256n$
Average length, $n = 3.5$	7168	2756	1792
Sign time	11.8 ms	11.9 ms	2.33 ms
Verify time	$0.27n$ ms	$0.30n$ ms	$2.77n$ ms

Table 1: Benchmark results. Let n be the number of signers. Results were computed on a laptop with a Core i3 processor at 2.4GHz and 2GB RAM, running Linux Ubuntu.

6 Implementation

In Section G we present technical arguments for how our schemes can be instantiated with RSA, and present a full specification of our schemes in Section H. To evaluate the viability of our schemes, we implemented the input-dependent-r version of our protocol as a module in OpenSSL [ope]. The code is available from [BGR11].

Overview of our implementation. We instantiate the permutation π with 2048-bit RSA with public exponent 65537, hash H with SHA-256, full-domain hash G with the industry-standard Mask Generating Function (MGF) using SHA-256 [RSA02], and the pseudorandom function PRF with HMAC-SHA-256 [BCK96]. Instead of hashing the permutation π as-is inside the hash function H, we replace it with a short fingerprint of the RSA public key computed using SHA-256. Thus, we have parameters $\ell_\pi = 2048, \ell_h = 256$, and $\ell_r = 128$; the ℓ_r value is per signer, and each signer also adds one bit of information to deal with the problem that RSA gives each signer a slightly different domain (see Section G). Therefore, the length of the aggregate signature for n signers is $2048 + 256 + 129n$ bits long (see Table 1). We now present a technical justification for this choice of parameters.

6.1 A sample set of parameter lengths

RSA modulus length. The choice of RSA modulus length is essentially orthogonal to the choice of all the other parameters, because it depends only on one's belief about the security of RSA. We will use 2048-bit modulus n, which NIST recommends for use until the year 2030 [BBB+07, Table 4]. Thus, ℓ_π is 2048 bits.

Choosing ℓ_f,ℓ_r and ℓ_h. Once the RSA modulus length is set, security depends on the term subtracted from ε in Theorem 4.2 (for the random-r version) or Theorem 5.1 (for the input-dependent-r version). We will aim for 128-bit security against adversarial running time. We bound the adversarial running time t in terms of the number of queries it issues, so that $t \geq q_S + q_G + q_H$.

We will also aim for 60-bit security against the chosen message attack against an honest signer. This number is lower than 2^{128}, because every signature has to be issued by the honest signer—thus, as opposed to hash values, an adversary cannot simply marshal more computational resources in order to get more signatures. We note that for an honest signer to issue 2^{60} signatures, assuming a public key is valid for one year, it would have to produce about 40 billion signatures per second. (Our benchmarks in Section 6 indicate that consumer-grade general-purpose PCs can produce about 100 signatures per second, so this builds in some safety margin.) Thus, we want to make sure the term subtracted from ε is at most about $t2^{-128} + q_S 2^{-60}$. In fact, for the truly-random-r version, we will do better: we will get close to $t2^{-128}$ as long as $q_S \leq 2^{60}$. For the input-dependent-r version, we will also do better when $q_S < 2^{60}$, because we will get close to $t2^{-128} + q_S^2 2^{-120}$. Given this goal, we choose the following parameters:

- Instantiate hash $H()$ with SHA-256 [SHA08] so ℓ_h is 256 bits.
- When using the random-r version, we have $\ell_{r,r}$ 192 bits.
- Since we use HMAC instantiated with $H()$, the PRF key has $\ell_k = \ell_h$ as 256 bits.
- When using the input-dependent-r version, we have $\ell_{r,p}$ is 128 bits.

Random-r version. Plugging these values (with $\ell_r = \ell_{r,r}$) into Theorem 4.2, we see that the term subtracted from ϵ is about

$$\begin{aligned}(q_S + q_H)(q_S + q_G + q_H)2^{-\ell_H} + q_S(q_S + q_H)2^{-\ell_r} + q_H^2 2^{3\ell_b - \ell_\pi} &\leq \\ 2^{128}(q_S + q_G + q_H)2^{-256} + 2^{60}(q_S + q_G + q_H)2^{-192} + 2^{128}(q_S + q_G + q_H)2^{-2045} &< \\ 2t2^{-128}\end{aligned}$$

as desired.

Input-dependent-r version. We assume that 256-bit HMAC provides 128-bit security against adversarial running time and 60-bit security against the number of queries as a pseudorandom function—that is, $\varepsilon_{\mathsf{PRF}}(q_S, t) \leq t2^{-128} + q_S 2^{-60}$. We plug these these values, along with $\ell_r = \ell_{r,p}$, into Theorem 4.2 to find

$$\begin{aligned}2(q_S + q_H)(q_S + q_G + q_H)2^{-\ell_H} + (q_G + q_H + q_S + (\ell_h + 2)q_S^2)2^{-\ell_r} + q_H^2 2^{3\ell_b - \ell_\pi} &\leq \\ 2 \cdot 2^{128}(q_S + q_G + q_H)2^{-256} + (q_G + q_H + q_S)2^{-128} + 258q_S^2 \cdot 2^{-128} + 2^{128}(q_S + q_G + q_H)2^{-2045} &< \\ 4t2^{-128} + 1.01q_S^2 2^{-120} &< \\ 4t2^{-128} + 1.01q_S 2^{-60}\end{aligned}$$

as desired.

In both versions, if the function PKFingerPrint does not explicitly output the public key but rather hashes it using SHA-256, then there will be an additional security loss of 2^{128}, because a collision of two key fingerprints will mean that the reduction R cannot deduce which π the forger F is using in its H-query.

7 Evaluation

Desiderata. We compare the performance of our scheme relative to other signatures that allow for lazy verification. Until this point, we have mainly been concerned with signature lengths. We now turn our attention to other practical considerations, and thus judge our schemes on the basis of (a) computation time, in addition to (b) signature lengths. Fast verify times are particularly crucial, since the i^{th} signer needs to verify $i-1$ times before it can sign once.

The competition. Table 1 compares our scheme to the 'trivial' solutions of using n RSA or ECDSA [Van92, IEE02] signatures, for roughly 128-bit security; namely the standard OpenSSL implementations of 256-bit ECDSA and 2048-bit RSA using SHA-256 and public exponent 65537. We focus on these two schemes because they are the current contenders for adoption in BGPsec [DHS].

While [BGLS03] is another potential solution with lazy verification, we do not benchmark it because no implementation was available to us, and implementing this scheme is outside the scope of this work. However, we do note that [BGLS03] offers the shortest bit lengths of all the schemes we consider. But, because [BGLS03] relies on bilinear pairing, this comes at the cost of less standard cryptographic assumptions and slower verification time. Verification in [BGLS03] requires one expensive bilinear pairing per signer[3]; as reported in [NNS10], highly optimized software on carefully chosen 256-bit curves would compute a pairing operation in about 2ms on hardware roughly comparable to ours (we note that these special curves and careful optimizations, not used in OpenSSL, could probably also improve the performance of ECDSA considerably).

Discussion. We can make the following observations:

1. ECDSA provides the shortest signature lengths of the three solutions when $n < 6$, while our scheme dominates the three for $n > 6$. However, if we consider BGPsec as a target application, it follows that average AS path lengths in the Internet tend to be around 3.5 hops long on average [COZ08]. Thus, ECDSA has the shortest signatures for the average case where $n = 3.5$. However, as n increases to $n > 6$, our scheme provides the shortest signatures; this could be advantageous for the weaker routers in the less well-connected portions of the Internet that tend to see longer AS-level paths with BGPsec.

2. Our scheme has computation time almost identical to trivial RSA. While ECSDA has the fastest signing time (3x faster), the *verification* times for RSA and our scheme are an order of magnitude faster than those of ECDSA, which is important when we consider BGPsec as a target application.

Thus, while there is no clear winner on all fronts, we conclude that our scheme enjoys the (a) more standard assumption of trapdoor permutations, and (b) fast verify times of RSA while (c) allowing for aggregate lengths that are comparable to those of ECDSA. We are currently analyzing the computation, communication, and storage impact of our schemes relative to RSA and ECDSA using empirical BGP routing data, with an eye towards assessing how the differences in signature length as well as signing/verifying times impact BGPsec performance.

Acknowledgements

We thank Anna Lysyanskaya for help with the early stages of this work, Hovav Shacham for helpful pointers, and the DHS S&T CSD Secure Routing project for many useful discussions that informed the design of our schemes. This work was supported by NSF Grants 1017907, 0546614, 0831281, 1012910, 1012798, and a gift from Cisco.

References

[BBB+07] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. *NIST Special Publication 800-57. Recommendation for KeyManagement – Part 1: General(Revised).* National Institute of Standards and Technology (NIST), March 2007. Available from `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[BGLS03] 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 416–32. Springer, 2003.

[3] A more efficient pairing-based scheme of [WM08] with a constant total number of pairings was shown insecure by [SVS+10].

[BGR11] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Implementation of sequential aggregate signatures with lazy verification, 2011. Available from `http://www.cs.bu.edu/fac/goldbe/papers/bgpsec-sigs.html`.

[BJ10] Ali Bagherzandi and Stanislaw Jarecki. Identity-based aggregate and multi-signature schemes based on rsa. In Phong Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 480–498. Springer, 2010.

[BNN07] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 411–422. Springer, 2007.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[CHKM10] Sanjit Chatterjee, Darrel Hankerson, Edward Knapp, and Alfred Menezes. Comparing two pairing-based aggregate signature schemes. *Des. Codes Cryptography*, 55(2-3):141–167, 2010.

[Cor02] Jean-Sébastian Coron. Optimal security proofs for PSS and other signature schemes. In Lars Knudsen, editor, *Advances in Cryptology—EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 28 April–2 May 2002.

[COZ08] Ying-Ju Chi, Ricardo Oliveira, and Lixia Zhang. Cyclops: The Internet AS-level observatory. *ACM SIGCOMM CCR*, 2008.

[CSC09] Saikat Chakrabarti 0002, Santosh Chandrasekhar, Mukesh Singhal, and Kenneth L. Calvert. An efficient and scalable quasi-aggregate signature scheme based on lfsr sequences. *IEEE Trans. Parallel Distrib. Syst.*, 20(7):1059–1072, 2009.

[DHS] Department of Homeland Security, Science and Technology Directorate, Cyber Security Division, Secure Protocols for Routing Infrastructure project. Personal Communication.

[DR02] Yevgeniy Dodis and Leonid Reyzin. On the power of claw-free permutations. In S. Cimato, C. Galdi, and G. Persiano, editors, *Third Conference on Security in Communication Networks SCN '02*, volume 2576 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2002.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[IEE02] IEEE Std 1363-2000. IEEE standard specifications for public-key cryptography, 2002.

[ITU02] *ITU-T Recommendation X.690: Information technology ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* Internation Telcommunication Union, 2002. Available at `http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. *IETF RFC 2104. HMAC: Keyed-Hashing for Message Authentication.* Internet Activities Board, February 1997. Available from `http://www.ietf.org/rfc/rfc2104.txt`.

[KLS00] S Kent, C Lynn, and K Seo. Secure border gateway protocol (S-BGP). *J. Selected Areas in Communications*, 18(4):582–592, April 2000.

[Lep11] M. Lepinski, editor. *BGPSEC Protocol Specification.* IETF Network Working Group, Internet-Draft, March 2011. Available from `http://tools.ietf.org/html/draft-lepinski-bgpsec-protocol-00`.

[LMRS04] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2004.

[LOS+06] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 465–485. Springer, 2006.

[Nev08] Gregory Neven. Efficient sequential aggregate signed data. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 52–69. Springer, 2008.

[NNS10] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2010.

[ope] Openssl toolkit. `http://openssl.org/`.

[RS09] Markus Rückert and Dominique Schröder. Aggregate and verifiably encrypted signatures from multilinear maps without random oracles. In Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-Hoon Kim, and Sang-Soo Yeo, editors, *ISA*, volume 5576 of *Lecture Notes in Computer Science*, pages 750–759. Springer, 2009.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[RSA02] *PKCS #1: RSA Encryption Standard. Version 2.1.* RSA Laboratories, June 2002. Available from `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`.

[SHA08] *FIPS Publication 180-3: Secure Hash Standard.* National Institute of Standards and Technology (NIST), October 2008. Available from `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[SVJR10] S. Sharmila Deva Selvi, S. Sree Vivek, J.Shriram, and C. Pandu Rangan. Efficient and provably secure identity based aggregate signature schemes with partial and full aggregation. Technical Report 2010/461, Cryptology ePrint archive, `http://eprint.iacr.org`, 2010.

[SVS+10] S. Sharmila Deva Selvi, S. Sree Vivek, J. Shriram, S. Kalaivani, and C. Pandu Rangan. Efficient and provably secure identity based aggregate signature schemes with partial and full aggregation. Technical Report 2009/290, Cryptology ePrint archive, `http://eprint.iacr.org`, 2010.

[Van92] Scott Vanstone. Responses to NIST's proposal. *Communications of the ACM*, 35:50–52, July 1992.

[WM08] Yiling Wen and Jianfeng Ma. An aggregate signature scheme with constant pairing operations. In *CSSE (3)*, pages 830–833. IEEE Computer Society, 2008.

[ZBD05] Huafei Zhu, Feng Bao, and Robert H. Deng. Sequential aggregate signatures working over independent homomorphic trapdoor one-way permutation domains. In Sihan Qing, Wenbo Mao, Javier Lopez, and Guilin Wang, editors, *ICICS*, volume 3783 of *Lecture Notes in Computer Science*, pages 207–219. Springer, 2005.

[ZSN05] Meiyuan Zhao, Sean W. Smith, and David M. Nicol. Aggregated path authentication for efficient bgp security. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *ACM Conference on Computer and Communications Security*, pages 128–138. ACM, 2005.

A Lazy verification and prior proposals

[LMRS04] is a sequential aggregate signature that produces a constant length aggregate x_i. Using the notation of Section 3, the signature algorithm requires the i^{th} signer to compute the aggregate x_i from the aggregate-so-far x_{i-1} as follows:

$$x_i = \pi_i^{-1}(x_{i-1} \oplus H(\pi_1, ..., \pi_i, m_1, ..., m_i)) \tag{4}$$

The security of [LMRS04] relies on the fact that the i^{th} signer verifies x_{i-1} before producing x_i as above. We now show how an adversary can forge a valid signature on a message $\widehat{m_i} \neq m_i$ by adversarially malforming the aggregate-so-far.

Suppose the adversary knows that the i^{th} signer is signing the message m_i. In that case, he sets

$$\begin{aligned}\widehat{x_{i-1}} = x_{i-1} &\oplus H(\pi_1, ..., \pi_i, m_1, ..., m_i) \\ &\oplus H(\pi_1, ..., \pi_i, m_1, ..., \widehat{m_i})\end{aligned}$$

where x_{i-1} is a valid signature on messages $m_1, ..., m_{i-1}$. If the i^{th} signer now produces a signature using the invalid aggregate-so-far $\widehat{x_{i-1}}$, then a little algebraic manipulation shows that the adversary now posses a valid signature on messages $m_1, ..., \widehat{m_i}$, and thus the security of [LMRS04] does not hold under lazy verification. The analogue of this attack also works on [Nev08]. Here, the adversary malforms only the hash value h_{i-1} as:

$$\widehat{h_{i-1}} = h_{i-1} \oplus H(\pi_i, m_i, x_{i-1}) \oplus H(\pi_i, \widehat{m_i}, x_{i-1})$$

B Definition of Sequential Aggregate Signature Schemes

As mentioned in Section 2, our definition is almost verbatim from [LMRS04], with differences to account for the fact that public keys of other signers are not input to the signing algorithm.

The adversary F is given a single public key and access to sequential aggregate signing oracle on the corresponding secret key. The advantage of F, $\mathsf{Adv\,AggSig}_F$, is defined to be its probability of success in the following game.

Setup. A key pair PK, SK is generated. The aggregate forger F is provided with PK, the challenge key.

Queries. F requests sequential aggregate signatures to be produced with SK on messages of its choice. For each query, it supplies an (alleged) sequential aggregate signature σ and an additional message m to be signed by the oracle under key SK. It can be adaptive, i.e., use the results of previous queries in order to decide on the current query.

Response. Finally, F outputs n distinct public keys $PK_1, \ldots, PK_n$ for some integer n of its choice. One of these keys must equal PK, the challenge key. Algorithm F also outputs messages $m_1, \ldots, m_n$, and a sequential aggregate signature σ.

The forger wins if the sequential aggregate signature σ is a valid sequential aggregate signature on messages $m_1, \ldots, m_n$ under keys $PK_1, \ldots, PK_n$, if $PK = PK_{i_*}$ for some $1 \leq i_* \leq n$, and if σ is nontrivial, i.e., F never issued a query on the message m_{i_*}. Note that i_* need not equal n: the forgery can be made in the middle of the sequence. The probability is over the coin tosses of the key-generation algorithm, the signing algorithm, and F.

In the random oracle model, F can also issue adaptive queries to the random oracle during its attack.

C Pseudorandom Functions

A pseudorandom function family (PRF) [GGM86] is one in which a randomly chosen function is indistinguishable from a truly random function by an observer of input-output behavior. We will consider only PRFs with variable input lengths and a fixed output length ℓ_r. The formal definition we need is as follows: if $\mathsf{PRF}_{\mathsf{seed}} : \{0,1\}^* \to \{0,1\}^{\ell_r}$ is a family of functions indexed by seed and D is an adversary that outputs a single bit (also known as *distinguisher*), consider the following two experiments. In the first, seed is chosen at random (not shown to D), and D gets to ask for outputs of $\mathsf{PRF}_{\mathsf{seed}}$ on inputs of its choice. In the second, a completely random function $f : \{0,1\}^* \to \{0,1\}^{\ell_r}$ is chosen at random, and D gets to ask for outputs of f on inputs of its choice. We will say the insecurity of PRF is the absolute value of the difference between the probabilities that D outputs 1 in the two experiments. We will denote by $\varepsilon_{\mathsf{PRF}}(q,t)$ the maximum insecurity of PRF against any D who asks at most q queries and runs in time t.

D Proof of Theorem 4.2

Our proof proceeds in two steps. First, we argue that with high probability forger F will not detect that it is interacting with the reduction R rather than the real oracles. To do so, we define the *view* of forger F to be everything F sees during a run, including F's input (the public key), F's private coinflips, and the answers F receives to its H, G, and signature queries. The reduction, unlike the real execution, may abort before F outputs its forgery, in which cases we call the view of F "aborted." F's inability to detect that it is interacting with the reduction R follows from the following two lemmas.

Lemma D.1 (Correct Simulation)**.** *Consider a view that forger F sees during the interaction with the reduction. Suppose it is not aborted. Then the probability that F sees that view when interacting with the reduction is the same as the probability F sees that view in the real execution.*

Lemma D.2 (Abort Probability)**.** *The reduction R aborts (either before or after the forger F outputs the forgery) with probability at most*

$$\Pr[ABORT] \le (q_S + q_H)(q_S + q_G + q_H)2^{-\ell_H} + q_S(q_S + q_H)2^{-\ell_r} + q_H^2 2^{-\ell_\pi} \tag{5}$$

Thus, if forger F outputs a forgery with probability ε when interacting with the real signer, then, by the union bound, the probability that it outputs a forgery and the reduction does not abort is at least $\varepsilon - \Pr[ABORT]$ (indeed, consider the union of two bad events: F doesn't output a forgery or the reduction aborts). But if the reduction does not abort, then it outputs a claw (x_n, z_n). By Lemma D.7, $\pi_*(x_n) = y_n$, where y_n is the value stored with the node N_n. And by construction of y_n on line 17 of Sim-H (Algorithm 6), $\rho_*(z_n) = y_n$. Hence, the reduction succeeds in finding a claw.

D.1 Proof of Correct Simulation Lemma D.1

The public key given to forger F and F's private coinflips are the same in the simulation and in the real execution. Thus, it remains to show that the reduction R correctly simulates H-queries, G-queries, and Sign-queries. Indeed, consider a not aborted view of F obtained during interaction with reduction R.

- ***H*-queries.** For every H-query in the view, the answer was placed into HT by Sim-H (Algorithm 6) or Sim-S (Algorithm 7). Each produced the HT entry by drawing an independent uniformly random η, just like the real execution random oracle. The probability of that particular answer is $2^{-\ell_H}$ in both cases.

- ***G*-queries.** Consider a G-query in the view. Sim-G responded to that G query by returning a value from the GT. Values are assigned to the GT by Sim-G,Sim-H, and Sim-S, so it suffices to show that each of these algorithms assigns a fresh uniformly random value to the GT, which would guarantee that the particular value seen in the view was assigned with probability $2^{-\ell_\pi}$, just like the real execution random oracle. First, Sim-G assigns values g to GT by choosing them uniformly at random (Algorithm 4). Next, Sim-H assigns the value $g(h_i) = \rho_*(z_i) \oplus x$ to the GT, where z_i is a fresh random value, and ρ_* is a permutation (Algorithm 6). Since the domain of ρ_* is equal to the range of G, and ρ_* is a permutation, it follows that $\rho_*(z_i)$ is uniformly random in the range of G. It therefore follows that g is a uniform random value. Similarly, Sim-S assigns the value $g(h') = \pi_*(x') \oplus x$, for a fresh random value x' and a permutation π_*, so $g(h')$ is also a uniform random value.

- **Sign-queries.** Observe that, given the answers to the relevant H and G queries and the input (m, h, x), there is a unique correct pair h', x' for every r. The value r appears in the view as the result of a uniformly random choice in both the simulated execution and the real one. The answers to the relevant H and G queries are also results of uniformly random choices in both cases, by the arguments made for H- and G-queries. And h', x' are the unique correct values in both cases.

D.2 Proof of Abort Probability Lemma D.2

It suffices to compute the probability that Lookup, Sim-H, Sim-G, Sim-S, and FindClaw abort and to add them up by union bound. Sim-G never aborts. The following four lemmas address the remaining four algorithms, in order.

Lemma D.3. *The probability that Lookup ever aborts during the whole execution is at most*

$$\frac{q_H^2}{2} 2^{-\ell_\pi} . \tag{6}$$

Proof. Lookup(x) aborts when the HTree contains two nodes, N_1 and N_2, such that a Lookup collision occurs, *i.e.,*

$$\pi_1(x) = y_1 \text{ and } \pi_2(x) = y_2 . \tag{7}$$

Because the size of the HTree is bounded by q_H, it suffices to show that, for every pair of nodes N_1, N_2, the probability that there exists x such that Equation 7 holds is at most $2^{-\ell_\pi}$. We will do so by proving two claims.

Claim D.4. *When a node N_i storing π_i, y_i is added to the HTree, then y_i is chosen uniformly at random and independent of all prior choices made in the interaction between the forger F and the reduction R.*

Proof. Recall that nodes are added to the HTree by Sim-H. From Sim-H, if $\pi_i = \pi_*$, then we can think of y_i as fresh random value, since $y_i \leftarrow \rho_*(z_i)$, where z_i is a fresh random value and ρ_* is a permutation. If $\pi_i \neq \pi_*$, then y_i is chosen in a slightly more complex manner. Note from Sim-H that $y_i \leftarrow$ Sim-G$(h_i) \oplus x$. Sim-G returns a uniform random value for each new input, and so it follows that y_i will be a fresh random value *as long as GT is not defined on h_i*. But if GT is defined on h_i, then Sim-H will abort, so N_i will not be added to the tree, anyway. □

Next, we need to show that the probability that there exists an x such that Equation 7 holds is small. To do this, we need to bound the probability that there are nodes N_1 and N_2 in HTree storing π_1, y_1 and π_2, y_2 such that $\pi_1(x_1) = \pi_2(x_2)$. In Section 4.2, we presented Claim 4.3 that showed this can occur with probability at most $2^{-\ell_\pi}$. The lemma follows by combining Claims D.4 and 4.3. □

Lemma D.5. *A single invocation of* Sim-H *aborts on line 10 with probability at most* $(q_S + q_G + q_H)2^{-\ell_H}$.

Proof. From Algorithm 6, we see that Sim-H aborts only if GT already stores some value for h_i. First observe that there are at most $q_G + q_S + q_H$ queries stored in the GT. (There are q_G G-queries, and every signing and H-query query adds at most one additional entry to GT.) Next, observe that $h_i = h_{i-1} \oplus \eta$ where η is a fresh random value. Thus, the probability that h_i collides with a value already in GT is bounded by $(q_G + q_S + q_H)2^{-\ell_H}$. □

Lemma D.6. *A single invocation of* Sim-S *aborts with probability at most* $(q_H + q_S)2^{-\ell_r} + (q_S + q_G + q_H)2^{-\ell_H}$.

Proof. There are two cases in which Sim-S aborts.

Abort due to H-collision. Sim-S on input (m, h, x) will abort on line 4 if it draws a random value r such that $Q = (\pi_*, m, r, x)$ exists in the HT. The number of entries in HT cannot exceed $q_H + q_S$, because those are the only queries that add entries (at most one each) to HT. Because r_i is a fresh random value, the probability it collides with one of these entries is bounded by $(q_H + q_S)2^{-\ell_r}$.

Abort due to G-collision. Sim-S will abort on line 14 if GT already stores some value for h'. The same argument as in Lemma D.5 shows that this happens with probability at most $(q_S + q_G + q_H)2^{-\ell_H}$. □

Lemma D.7. *The probability of abort on line 3 of* FindClaw *is at most* $\frac{q_H^2}{2}2^{-\ell_\pi}$*. And if the abort does not happen, then* $\pi_*(x_n) = y_n$*, where* y_n *is stored with* N_n*.*

Proof. Suppose the forger F outputs a forgery $\sigma = (\vec{\pi_n}, \vec{m_n}, \vec{r_n}, h_n, x_n)$ with $\pi_n = \pi_*$ that is valid relative to HT, GT, which means that FindClaw is invoked.

Consider running $\mathsf{Ver}^{\mathsf{HT},\mathsf{GT}}(\sigma)$ (Algorithm 2) to verify the forgery σ. The verification algorithm asks a sequence of H-queries $Q_n, ..., Q_1$, where $Q_1 = (\pi_1, r_1, m_1, \epsilon)$ and $Q_i = (\pi_i, m_i, r_i, x_{i-1})$ for every $i = 2...n$. We know that all these queries have been asked by forger F and are therefore in HT. Let $\eta_n, \ldots, \eta_1$ be the answers to these queries. Note that these queries could not have been placed into HT by Sim-S, because m_n is different from every message queried to Sim-S with π_*. Thus, they were asked by the forger to Sim-H.

The verification algorithm also asks a sequence of G-queries $h_n, \ldots, h_1$, where $h_{n-1} = h_n \oplus \eta_{n-1}$, $h_{n-2} = h_{n-1} \oplus \eta_{n-1}, \ldots, h_1 = h_2 \oplus \eta_2$. Because the forgery is valid, $h_1 = \eta_1$, and therefore $h_i = \bigoplus_{j=1}^{i} \eta_j$. Note that all these G queries are in GT.

Note that Q_1 is tethered to the root of the HTree by Definition 4.1, and will therefore be placed in the HTree by Sim-H with the value $h_1 = \eta_1$. Because the forgery is valid, the x_1 value in Q_2 must satisfy $\pi_1(x_1) = y_1$, where $y_1 = G(h_1)$. Thus, Q_2 is tethered to Q_1. That does not necessarily mean that Q_2 itself is in the HTree. However, if it is, then it has the values $h_2 = h_1 \oplus \eta_2$ and $y_2 = G(h_2 \oplus x_1)$ stored in it. Thus, if Q_2 is in the HTree, then Q_3 is tethered to Q_2, because x_2 in Q_3 must satisfy $\pi_2(x_2) = y_2$, because that condition on Q_3 is necessary in order for the verification algorithm to query Q_2. Similarly, if Q_3 is in the HTree, then Q_4 must be tethered to it. By induction, either there exists $i > 1$ such that Q_i is tethered to Q_{i-1} but is not in the HTree, or all $Q_1, \ldots, Q_n$ are in the HTree. In the latter case, $\mathsf{Lookup}(x_n)$, if it does not abort, will return the node for the query Q_n (because Q_n was asked during verification, which happens only if $\pi_n(x_n) = y_n$), and thus FindClaw will not abort.

Thus, we have shown that, if FindClaw doesn't abort, then $\pi_n(x_n) = y_n$ (note that $\pi_n = \pi_*$), and that FindClaw will never abort unless there exists a query Q that was asked to Sim-H, is tethered to another query in HTree, but is not in the HTree. In Section 4.2, we presented Claim 4.4 that shows that this occurs with probability most $q'_H 2^{-\ell_\pi}$, where q'_H is the number of H queries made after query Q. To bound the probability that a tethered H-query exists outside the HTree, we add up over all q_H queries, to obtain $\frac{q_H^2}{2} 2^{-\ell_\pi}$ by the union bound. □

Finally, Sim-H is called at most q_H times, Sim-S is called at most q_S times, FindClaw is called once and thus Lemma D.2 holds by a union bound. .

E Proof of Theorem 5.1

We present only on the parts of the proof that are different from Theorem 4.2.

Changes to the reduction R. We will assume that forger F never makes the same signature query twice, because it would get the same result, anyway (formally, we can always modify F not to ask the same signature query twice by keeping a table of previously requested signature queries). Reduction R uses the same algorithms as before, except that we replace the ABORT on line 4 of Sim-S (Algorithm 7) with $\eta \leftarrow \mathsf{HT}(Q)$.

Changes to Lemma D.1. Because Line 1 of Sim-S uses a truly random rather than a pseudorandom r, the probabilities of non-aborting views are no longer the same. However, by a standard reduction to the security of the PRF, the probability that forger F produces a forgery from a nonaborting view in the simulation must be at least $\varepsilon - \varepsilon_{\mathsf{PRF}}$.

Changes to Lemma D.2. This lemma changes only in Lemma D.6. The new version is:

Lemma E.1. *The probability that, during any of the q_S queries, Sim-S aborts is at most*

$$q_S(q_S + q_G + q_H)2^{-\ell_H} + (q_S + q_H)^2 2^{-\ell_H} + (q_G + q_H + q_S + (\ell_h + 2)q_S^2)2^{-\ell_r}\,.$$

Proof. When does the modified Sim-S abort? When $h' = \eta \oplus h$ is in GT. There are two cases. First, if η (and thus h') is a fresh random value, then the same argument used in Lemma D.6 holds, so abort probability is at most $(q_S + q_G + q_H)2^{-\ell_H}$.

However, we must now consider a new case where Sim-S aborts; namely, if η is *not* a fresh random value. η will not be a fresh random value if Sim-S is given a sign query (π_*, m, x, h) and draws an r that defines a query $Q = (\pi_*, m, r, x)$ that is (a) already in HT *and* (b) the value $h' = \eta \oplus h$ is already stored in the GT (where recall that $\eta = \mathsf{HT}(Q)$ and h was given as part of the query to Sim-S). Call such r *bad* for the sign-query's m, h, x. In Claim 5.2 we argued that Sim-S draws a bad r with probability at most $(q_G + q_H + q_S + (\ell_h + 2)q_S^2)2^{-\ell_r}$ □

F Combinatorial interlude: A proof of Theorem 5.3

Here we solve the combinatorial problem of Section 5. We start with a prelude problem:

A Prelude Problem. Suppose β values $\eta_1, \ldots, \eta_\beta$ are chosen uniformly at random as ℓ_H-bit strings and the $\beta \times \beta$ matrix θ is computed as $\theta_{ij} = \eta_i \oplus \eta_j$. The diagonal of θ has all zero entries. Can we bound the size C_θ of the biggest nonzero collision within θ?

Lemma F.1. *With probability at least $1 - \beta^2 2^{-\ell_H}$, all the η_j values are distinct and $C_\theta \le 2\ell_h + 4$.*

Proof. Since we are not considering the 0 collision, we can remove the diagonal from our consideration and, in fact, focus only on the upper triangle of elements above the diagonal (the elements below the diagonal are equal to them, so we will get a nonzero collision of size $2k$ in θ if and only if we have k elements colliding in the upper triangle). Note that entries in a given row or given column are always distinct, unless $\eta_i = \eta_j$ for some $i \neq j$, which happens with probability no more than $\frac{\beta^2}{2}2^{-\ell_H}$. Consider the event that there is a collision of size k in the upper triangle or the η_j values are not distinct; let p_k be its probability. We can consider all subsets of k entries of the upper triangle in two parts: those subsets in which at least two elements share a row or a column (which, taken altogether, are covered by the case of nondistinct η_j values), and those in which all columns are distinct and all rows are distinct. Taking a union bound over all k-element subsets then gives us

$$p_k \leq \frac{\beta^2}{2}2^{-\ell_H} + \sum_{\substack{0 \leq j_1 < \cdots < j_k \leq \beta; \\ \text{distinct } i_1 < j_1, \ldots, i_k < j_k}} \Pr[\eta_{i_1} \oplus \eta_{j_1} = \cdots = \eta_{i_k} \oplus \eta_{j_k}]\,. \tag{8}$$

The constraint $i_1 < j_1, \ldots, i_k < j_k$ comes from the fact that we are only considering the upper triangle. Because, for every a, the index j_a is greater than $j_1, \ldots, j_{a-1}$ and therefore also greater than $i_1, \ldots, i_a$, and the value η_{j_a} was chosen uniformly at random, we get that the value η_{j_a} is independent of $\eta_{i_1} \oplus \eta_{j_1} = \cdots = \eta_{i_{a-1}} \oplus \eta_{j_{a-1}}$, and of η_{i_a}, and therefore the a^{th} element $\eta_{i_a} \oplus \eta_{j_a}$ of the subset is independent of all the previous elements of the subset. This implies that (subject to the distinctness requirement on the i_as and j_as), $\Pr[\eta_{i_1} \oplus \eta_{j_1} = \cdots = \eta_{i_k} \oplus \eta_{j_k}] = 2^{-\ell_H(k-1)}$.

We have thus bounded the probability that a given k-element subset with distinct rows and distinct columns is a collision. That is, we bounded each addend of the sum in Equation 8. How many such subsets are there? There are β rows, β columns, and we are choosing k distinct rows and k distinct columns, so there are at most $\binom{2\beta}{2k}$ of them. Thus, the sum has at most $\binom{2\beta}{2k}$ addends. Substituting into the above formula, we get

$$\begin{aligned} p_k &\leq \frac{\beta^2}{2}2^{-\ell_H} + \binom{2\beta}{2k}2^{-\ell_H(k-1)} \\ &\leq \frac{\beta^2}{2^{\ell_H+1}} + \left(\frac{\beta e}{k}\right)^{2k} 2^{-\ell_H(k-1)} \\ &= \frac{\beta^2}{2^{\ell_H+1}} + \left(\frac{\beta^2 e^2}{k^2 2^{\ell_H}}\right)^{k-1} \frac{\beta^2 e^2}{k^2} \\ &= \frac{\beta^2}{2^{\ell_H+1}} \left(1 + \left(\frac{\beta^2 e^2}{k^2 2^{\ell_H}}\right)^{k-1} \frac{e^2 2^{\ell_H+1}}{k^2}\right) \end{aligned}$$

Observe that we can assume $\beta^2/2^{\ell_H} < 1$ (otherwise, the statement of the lemma is vacuous). So if $k \geq 4$, then $k^2 > 2e^2$ and thus $\frac{\beta^2 e^2}{k^2 2^{\ell_H}} < \frac{1}{2}$. So set $k = \ell_H + 2 \geq 4$. We get

$$p_k < \frac{\beta^2}{2^{\ell_H+1}} \left(1 + \left(\frac{1}{2}\right)^{\ell_H+1} \frac{e^2 2^{\ell_H+1}}{k^2}\right) < \frac{\beta^2}{2^{\ell_H+1}} \cdot 2 = \frac{\beta^2}{2^{\ell_H}}\,.$$

Thus we have that with probability $< \frac{\beta^2}{2^{\ell_H}}$, we have a collision in the upper triangle of size $\ell_H + 2$. The lemma follows because of the symmetry of the matrix. □

We are now ready to solve the combinatorial problem of Section 5:

Proof of Theorem 5.3. Assume the event of Lemma F.1 happens (it happens with probability $1 - \beta^2 2^{-\ell_H}$). Consider the largest collision in ζ; suppose its size is c_1 and its value is v_1. It has all distinct rows (because η_j values are distinct by the assumption that Lemma F.1 holds). Therefore, without loss of generality, we can assume that the collision occurs in rows $1, \ldots, c_1$ of ζ (this is just for convenience of notation). Call rows $1, \ldots, c_1$ the first *layer* of ζ. Consider the ith row of the first layer and the entry in that row that participates in the collision. That entry has value v_1, and therefore $h'_i = \eta_j \oplus v_1$ for some η_j. Thus, each of the values $h'_1, \ldots, h'_{c_1}$ is simply some η value shifted by v_1, and therefore the first layer of ζ corresponds to some c_1 distinct rows of the matrix θ from the prelude problem (distinct because the problem statement requires the h' values to be distinct), except with v_1 added to all values. Therefore, every collision that does *not* have value v_1 in the first layer of ζ is also a nonzero collision in θ; by the assumption that the event of Lemma F.1 holds, it has size at most C_θ.

Consider now the second largest collision in ζ, of size c_2 and value v_2. By the same argument as before, it has c_2 distinct rows. This time, some of these rows may be in the *first* layer of ζ; however, there are no than C_θ such rows, because the first layer of ζ has no collisions with value not equal to v_1 of size greater than C_θ. Let c'_2 be the remaining rows; we know $c_2 \leq c'_2 + C_\theta$ and can assume without loss of generality that these rows are $c_1 + 1, \ldots, c_1 + c'_2$ (again, this is just for convenience of notation). Call these rows the *second layer* of ζ. Suppose some other collisions occur in the second layer of ζ. Using the same argument we made for the first layer, it follows that these collisions correspond to nonzero collisions in θ (obtained by adding v_2 to all the values) and thus have size at most C_θ.

In general, if we consider the ith largest collision in ζ of size c_i, up to $C_\theta(i-1)$ of its rows can be from previous layers (*i.e.,* up to C_θ from each of the previous layers). Let c'_i be the remaining rows (call them the i^{th} layer); we have $c_i \leq c'_i + C_\theta(i-1)$. No other collision in the i^{th} layer is of size more than C_θ.

Thus, the total size of γ collisions is at most $c_1 + \cdots + c_\gamma \leq c_1 + c'_2 + \cdots + c'_\gamma + C_\theta(1+2+\cdots+\gamma-1)$. Because $c_1, c'_2, \ldots, c'_\gamma$ refer to sizes of nonoverlapping layers, their sum is at most α. The theorem follows by observing that $1 + 2 + \cdots + \gamma - 1 < \gamma^2/2$ and by substituting $C_\theta \leq \ell_h + 2$ from Lemma F.1. $\square$

G Handling Permutations with Different Domains, such as RSA

Similarly to the schemes of [LMRS04] and [Nev08], our scheme extends to the case when each signer's permutation has its own domain, as long as no domain is much larger than the intersection of all the domains. For instance, if we instantiate our scheme with RSA using 2048-bit moduli, then each signer's permutation domain will be a subset of $\{0,1\}^{2048}$. The intersection of all the domains, however, will be at least the set of all 2048-bit strings that begin with 0, and thus no domain is more than twice the intersection of all the domains. (Following ideas of Zhu, Bao and Deng [ZBD05], the scheme can also be generalized to the case of domains of very different sizes, such as when different signers use RSA moduli of different lengths; we do not present this generalization here, because we expect all the moduli to be of the same standardized length in a typical deployment.)

Changes to the Scheme To explain how we modify the scheme, we need to fix some notation. We will assume that the domains of all permutations are subsets of $\{0,1\}^{\ell_\pi}$, and that the intersection of all the domains contains some set D closed under $\oplus$ (recall that the operation does not have to be exclusive-or—any group operation over D will do). Furthermore, we will assume that there is an efficient and efficiently invertible bijection SPLIT that takes an element X of $\{0,1\}^{\ell_\pi}$ and produces two values b, x, with $x \in D$ and $b \in \{0,1\}^{\ell_b}$ with ℓ_b close to $\ell_\pi - \log_2 |D|$. (For the case of RSA described above, $\ell_b = 1$. The function SPLIT sets $b = 1, x = X - 2^{2047}$ if $X \geq 2^{2047}$, and $b = 0, x = X$ otherwise.)

We will change G to output elements of D instead of $\{0,1\}^{\ell_\pi}$. We will change Step 6 of the signing algorithm (Algorithm 1) as follows:

$$X_i \leftarrow \pi_i^{-1}(y_i); (b_i, x_i) \leftarrow \mathrm{SPLIT}(X_i)$$

The signing algorithm will output b_i in addition to x_i. The entire vector $\vec{b_n}$ will be input to the verifying algorithm (Algorithm 2), which will be modified as follows: Step 2 will be replaced with

$$X_i \leftarrow \mathrm{SPLIT}^{-1}(b_i, x_i); y_i \leftarrow \pi_i(X_i)\,.$$

Changes to the Reduction The security reduction needs to modified as follows. Recall that the reduction relies on the HTree, which is built up so that a child node is always tethered to a parent node. We will change the definition of "tethered" (Definition 4.1): an H-query Q containing $x \neq \epsilon$ will be *tethered* to some node N_i in the HTree if that node contains π_i, y_i such that there exists $b \in \{0,1\}^{\ell_b}$ for which $\pi_i(\mathrm{SPLIT}^{-1}(b, x)) = y_i$.

Lookup (Algorithm 3) and FindClaw (Algorithm 5) will need to try all possible values of b to combine with the given x in order to find X to which π can be applied. Thus Step 4 of Lookup becomes

$$\text{Nodelist} = \{\text{all nodes } N \text{ in HTree containing } \pi, y \text{ such that } \exists b \in \{0,1\}^{\ell_b} \text{ such that } \pi(\mathrm{SPLIT}^{-1}(b, x)) = y\}\,.$$

Step 5 of FindClaw becomes

Find $b \in \{0,1\}^{\ell_b}$ such that $\pi_*(\mathrm{SPLIT}^{-1}(b, x_n)) = \rho_*(z_n)\}$;
return claw $(\mathrm{SPLIT}^{-1}(b, x_n), z_n)$.

Sim-H and Sim-S (Algorithms 6 and 7) need to search for y_i and y', respectively, that are in D. Thus, Steps 16 and 17 in Sim-H need to be repeated until $y_i \in D$ (also, z_i should be drawn from Domain(π_*) rather than $\{0,1\}^{\ell_\pi}$). Similarly, Steps 9 and 10 need to repeatedly draw $X' \overset{R}{\leftarrow}$ Domain(π_*) until $y' = \pi_*(X')$ is in D; the output of Sim-S should include $(b', x') = \mathrm{SPLIT}(X')$.

Finally, Step 2 of Sim-G (Algorithm 4) should draw g from D rather than $\{0,1\}^{\ell_\pi}$.

Changes to the Analysis The above changes to the reduction will cause it to run 2^{ℓ_b} times slower (thus, twice as slow for the RSA example).

The analysis undergoes the following changes. Lemma D.1 remains true, but the proof is a bit more delicate: when arguing about the correct simulation of G queries, we need to rely on the fact that the new procedures in Sim-H and Sim-S still produce an output for G that is uniform in D, because y_i in Sim-H and y' in Sim-S are uniform in D, because they are produced by sampling a uniform distribution until an element of D is found. Claim D.4 also remains true, using the same argument.

Equation 7 and Claim 4.3 change as follows.

Claim G.1. *Given two functions π_1, π_2 whose domains are subsets of $\{0,1\}^{\ell_\pi} = \mathrm{SPLIT}^{-1}(\{0,1\}^{\ell_b} \times D)$ and two uniformly chosen random values y_1, y_2 in D, the probability that there exists x, b_1, b_2 such that*

$$\pi_1(\mathrm{SPLIT}^{-1}(b_1, x)) = y_1 \text{ and } \pi_2(\mathrm{SPLIT}^{-1}(b_2, x)) = y_2 \tag{9}$$

holds is at most $2^{3\ell_b - \ell_\pi}$.

Proof. Define the set of preimages of y_1 as $S_{y_1} = \{(b_1, x) \text{ such that } \pi_1(\mathrm{SPLIT}^{-1}(b_1, x)) = y_1\}$. Suppose $|S_{y_1}| = \alpha$. Then there are at most $2^{\ell_b} \cdot \alpha$ choices of y_2 for which there exist x, b_1, b_2 satisfying Equation 9, because each triple in (x, b_1, b_2) with $b_2 \in \{0,1\}^\ell b$ and $(b_1, x) \in S_{y_1}$ gives rise to at most one $y_2 = \pi_2(\text{SPLIT}^{-1}(b_2, x))$. Because y_2 is chosen uniformly from a set of size $|D|$, the probability that x, b_1, b_2 satisfying Equation 9 exist is at most $\frac{2^{\ell_b} \cdot \alpha}{|D|}$. Thus, the desired probability is at most

$$\sum_{\alpha} \frac{2^{\ell_b} \cdot \alpha}{|D|} \Pr_{y_1}[|S_{y_1}| = \alpha] = \frac{2^{\ell_b}}{|D|} \sum_{\alpha} \alpha \cdot \frac{|\{y_1 \text{ s.t. } |S_{y_1}| = \alpha\}|}{|D|}.$$

Observing that $\sum_{\alpha} \alpha \cdot |\{y_1 \text{ s.t. } |S_{y_1}| = \alpha\}| = \sum_{y_1} |S_{y_1}| = |\text{Domain}(\pi_1)|$, and that $|\text{Domain}(\pi_1)|/|D| \leq 2^{\ell_b}$, we get that the probability is at most $2^{2\ell_b}/|D|$. Further observing that $|D| \geq 2^{\ell_\pi - \ell_b}$, we get the desired bound. $\square$

This change results in the corresponding change in Lemma D.3: the $\frac{q_H^2}{2} 2^{-\ell_\pi}$ probability gets replaced by $\frac{q_H^2}{2} 2^{3\ell_b - \ell_\pi}$.

Finally, Claim 4.4 needs modification. Equation 2 gets replaced with

$$\exists b \in \{0,1\}^{\ell_b} \text{ such that } \pi'(\text{SPLIT}^{-1}(b, x)) = y',$$

which is satisfied with probability $2^{\ell_b - \ell_\pi}$. Thus, the probability in the statement of Claim 4.4 changes to $q'_H 2^{\ell_b - \ell_\pi}$ and the probability that FindClaw aborts (bounded in Lemma D.7) changes to $\frac{q_H^2}{2} 2^{\ell_b - \ell_\pi}$

The above changes result in the the $q_H 2^{-\ell_\pi}$ term in the formulas of Lemma D.2 and Theorems 4.2 and 5.1 being replaced with $q_H 2^{3\ell_b - \ell_\pi}$. Because ℓ_b is much smaller than ℓ_π, this change has no material impact on the security of the scheme.

H A Detailed Specification of the Signature Scheme

In this section we instantiate our signature scheme with RSA and describe implementation details. The following issues, in particular, need to be decided.

- How to encode multiple arguments to a hash function into a single string? We use simple concatenation; to provide for unique parsing (which is needed for the security proof), we ensure that there is at most one variable-length value in the concatenation.
- How to encode the empty string ϵ? We use zero strings of the appropriate length, because it causes no difficulty for our scheme.
- How to convert integers to bit strings and back? We use standardized conversion routines.
- What to use for the long-output hash function G? We use an industry-standard "mask generation function."
- How to represent the permutation π_i as input to the hash function? Any fixed-length encoding than unambiguously specifies the RSA modulus N and exponent E will be fine. However, there may be situations in which a more convenient public-key "fingerprint"—such as the hash value of the certificate—may be more readily available. Such a fingerprint is also acceptable, as long as it is computed by applying some cryptographic hash function to an unambiguous encoding of the public key (in the proof, the hash function would have to be modeled as a random oracle, and the reduction would know the value of the public key by looking at queries to this hash function).

We simultaneously specify two versions of our scheme: the version with random r of length $\ell_{r,r}$ (as described in Section 3) and the version with shorter input-dependent pseudorandom r of length $\ell_{r,p}$ (as described in Section 5). Because most the steps are the same, we combine the two versions into one specification.

H.1 Parameters, notations, and primitives

We specify our protocol in terms of bit strings; lengths are bit lengths. However, if all lengths are multiples of eight, then the protocol can be implemented in terms of octet strings; the specification does not convert lengths that are multiples of eight into lengths that are not multiples of eight. The following bit lengths are fixed:

- Length ℓ_h of the hash output
- Length ℓ_π of the RSA modulus
- Length $\ell_b = 1$ (see Section G)
- Length $\ell_{r,r}$ of the per-signature randomness r (needed only if the random-r version of the protocol is used—see Section 3)
- Length ℓ_k of a pseudorandom function key (needed only if the input-dependent-r version of the protocol is used—see Section 5)
- Length $\ell_{r,p}$ of the pseudorandom function output (needed only if the input-dependent-r version of the protocol is used—see Section 5)
- Length ℓ_f of the fingerprint (as discussed in the last bullet of this Section, $\ell_f \geq \ell_h$)

We provide some guidelines on choosing these parameters in Section 6.1. Notation and primitives used in our specification are as follows:

- The notation $\oplus$ is the bitwise XOR function.
- The notation $x||y$ means x concatenated with y.
- BS2IP(m) converts bit strings to non-negative integers. For example, see Section 5.5.1 of IEEE Std 1363-2000 [IEE02]. We use bit strings rather than octet strings only for notational convenience; if octet strings are preferred, the primitive may be instantiated using the OS2IP(m) primitive from PKCS #1 v2.1 [RSA02] or, equivalently, [IEE02, Section 5.5.3].
- I2BSP(m, ℓ) converts integers between 0 and $2^\ell - 1$ to bit string of length ℓ. For example, see Section 5.5.1 of IEEE Std 1363-2000 [IEE02]. If octet strings are preferred and ℓ is a multiple of 8, this primitive may be instantiated using the OS2IP(m) primitive from PKCS #1 v2.1 [RSA02] or, equivalently, [IEE02, Section 5.5.3].
- RAND(ℓ) is a cryptographically-strong random bit generator that generates ℓ fresh random bits each time it is invoked. Implementation of RAND() is system-dependent. In some systems, it may be implemented using pseudorandom bit generation.
- $H(m)$ is a cryptographic hash function that takes in variable length strings m and produces outputs of bit length ℓ_h.

- $\mathsf{PRF}(k, m)$ is a pseudorandom function taking in a key k of length ℓ_k and an input m of variable length, and producing an output of length $\ell_{r,p}$.

  The $\mathsf{PRF}(k, m)$ may be instantiated with the function HMAC(k, m), where HMAC is as specified in RFC 2104 [KBC97] (see also [BCK96]). If HMAC is implemented with $H()$ as the cryptographic hash function, then key k should have length at least ℓ_h, and the output of HMAC(k, m) will be of length ℓ_h, which should then be truncated to $\ell_{r,p}$ bits.

- MGF(m) is a mask generation function, taking in an input m of variable length, and produce an output bit string of length ℓ_π bits. MGF() may be implemented using the MGF1 algorithm (see Appendix B of PKCS #1 v2.1 [RSA02]) using $H()$ as the cryptographic hash function.

- RSAGEN(ℓ_π) is any secure RSA Key Generation function that produces an *integer* public exponent e_i, an *integer* RSA modulus N_i of length ℓ_π such that the RSA modulus $N_i \in [2^{\ell_\pi - 1}, 2^{\ell_\pi} - 1]$, as well as a corresponding RSA secret key $RSASK$. For example, see [IEE02, Appendix A.16.11]. Note that we do not specify the exact format of the secret key; in particular, either of the formats in [IEE02] is fine.

- RSASP1$(RSASK, m)$ is the RSA signature primitive from PKCS #1 v2.1 [RSA02] that takes in an RSA secret key $RSASK$, and a integer message m in the range $[0, n-1]$, and produces a integer s in the range $[0, n-1]$. Recall that n is the integer RSA modulus obtained from RSAGEN.

- RSAVP1$((N, E), m)$ is the RSA verification primitive from PKCS #1 v1.2 [RSA02], that takes in an integer RSA public key (N, E), and an integer message m in the range $[0, N-1]$ and produces a integer in the range $[0, N-1]$. Note that N is the integer RSA modulus obtained from RSAGEN.

- PKFingerPrint(PK_i) produces a length ℓ_f bit string which is a fingerprint of the public key $PK_i = (N_i, E_i)$ obtained from RSAGEN. This may be any fixed-length encoding from which N_i and E_i can be unambiguously obtained, such as the DER encoding [ITU02] of PK_i, or the string I2BSP(n, ℓ_π)||I2BSP(e, ℓ_π) (in which case $\ell_f = 2\ell_\pi$). It may also be the cryptographic hash function H applied to any unambiguous (not necessarily fixed-length) encoding of N, E. For example, a certificate fingerprint computed with the hash function H will work. Whatever option is chosen, it is important that the length ℓ_f should be fixed and at least ℓ_h.

H.2 Algorithms

Key generation, signing, and verifying are specified in Algorithms 8, 9, and 10. There are switches in the algorithms to indicate which version (random-r or input-dependent-r) is being run. To help with implementation, we also specify the types and/or bit lengths for each value of the protocol. We do not specify how to encode the keys and the signatures that are output by our algorithms, as this choice depends on the interoperability requirements of the application. DER encoding [ITU02] is one possible choice; simpler encodings—for instance, concatenation when all the lengths are unambiguously known—may also be a valid choice.

Algorithm 8 Gen: Key generation algorithm for the i^{th} Signer

```
1: Let $((N_i, E_i), RSASK_i) = RSAGEN(\ell_\pi)$. This produces an RSA modulus $N_i$ which is an
   integer in the range $[0, 2^\pi - 1]$, a integer public exponent $E_i$, and an RSA secret key $SKRSA_i$.
2: Ensure that the RSA modulus $N_i \in [2^{\ell_\pi - 1}, 2^{\ell_\pi} - 1])$.
3: if Input-Dependent-$r$ Version then
4:    Let $k_i$ be a random bit string of length $\ell_k$, obtained by running RAND($\ell_k$).
5:    return Public Key $PK_i = (N_i, E_i)$ and Secret Key $SK_i = (RSASK_i, k_i)$.
6: else
7:    return Public Key $PK_i = (N_i, E_i)$ and Secret Key $SK_i = RSASK_i$.
```

Algorithm 9 Sign: The i^{th} Signer's algorithm.

```
Require: $PK_i, SK_i$, the key of the $i^{th}$ signer. These should be valid keys obtained from the Gen
   algorithm.
Require: $f_i$ where $f_i$ is the fingerprint of $PK_i$. This should be a valid fingerprint obtained from
   PKFingerPrint($PK_i$) and should be a bit string of length $\ell_f$.
Require: $m_i$ the message bit string to be signed by the $i^{th}$ signer
Require: $x_{i-1}, h_{i-1}$ are obtained from the $i - 1^{th}$ signer, if $i > 1$. If $i = 1$, then $x_{i-1}$ is the string
   of $\ell_\pi$ zero bits and $h_{i-1}$ is the string of $\ell_h$ zero bits.
1: Check that $x_{i-1}$ is a bit string of length $\ell_\pi$ and $h_{i-1}$ is a bit string of length $\ell_h$. If not, output
   "Sign Fail" and exit.
2: if Input-Dependent-$r$ Version then
3:    Let $r_i = PRF(k_i, m_i||x_{i-1}||h_{i-1})$. This produces an output bit string of length $\ell_{r,p}$ bits.
4: else
5:    Let $r_i = \mathrm{RAND}(\ell_{r,r})$. This produces an output bit string of length $\ell_{r,r}$ bits.
6: Let $\eta_i = H(f_i||m_i||x_{i-1}||r_i)$. This produces an output bit string of length $\ell_h$.
7: Let $h_i = h_{i-1} \oplus \eta_i$. This produces an output bit string of length $\ell_h$.
8: Let $g_i = MGF(h_i)$. This produces a bit string of length $\ell_\pi$.
9: Let $y_i = g_i \oplus x_{i-1}$. This produces a bit string of length $\ell_\pi$.
10: Set the most significant bit of $y_i$ to 0.
11: Let $Y_i = \mathrm{BS2IP}(y_i)$. This produces an integer in the range $[0, N_i - 1]$ (more precisely, this
    integer lies in the smaller range $[0, 2^{\ell_\pi - 1} - 1]$).
12: Let $X_i = \mathrm{RSASIG}(SK_i, y_i)$. This produces an integer in the range $[0, N_i - 1]$
13: Let $x_i = \mathrm{I2BSP}(X_i, \ell_\pi)$. This produces an output bit string of length $\ell_\pi$.
14: Let $b_i$ be the most significant bit of $x_i$.
15: Set the most significant bit of $x_i$ to 0.
16: return $m_i, r_i, b_i, x_i, h_i$
```

Algorithm 10 Ver: The Verification Algorithm

```
Require: $\vec{PK}_n$ (a vector of $n$ RSA public keys), $\vec{m_n}$ (a vector of $n$ messages), $\vec{r_n}$ (a vector of $n$
    strings of length $\ell_{r,r}$ in the random-$r$ version or $\ell_{r,p}$ in the input-dependent-$r$ version), $\vec{b_n}$ (a
    vector of $n$ bits), $x_n$ (a bit string of length $\ell_\pi$ bits), $h_n$ (a bit string of length $\ell_h$ bits)
 1: for $i = n, n-1, ...., 1$ do
 2:   Set the most significant bit of $x_i$ to $b_i$.
 3:   Let $X_i = \text{BS2IP}(x_i)$.
 4:   if $X_i \geq N_i$ then
 5:     return 0
 6:   Let $Y_i = RSAVER(PK_i, X_i)$. This produces an integer between 0 and $N_i - 1$.
 7:   Let $y_i = \text{I2BSP}(Y_i, \ell_\pi)$. This produces an output of length $\ell_\pi$.
 8:   Let $g_i = MGF(h_i)$. This produces an output of length $\ell_\pi$.
 9:   Let $x_{i-1} = g_i \oplus y_i$. Set the most significant bit of $x_{i-1}$ to 0.
10:   Let $\eta_i = H(f_i||m_i||x_{i-1}||r_i)$. This produces an output of length $\ell_h$.
11:   Let $h_{i-1} = h_i \oplus \eta_i$. This produces an output of length $\ell_h$.
12: if $h_0$ is the string of $\ell_h$ zero bits and $x_0$ is the string of $\ell_\pi$ zero bits then
13:   return 1
14: else
15:   return 0
```