Synchronized Aggregate Signatures from the RSA Assumption

Susan Hohenberger[*]
Johns Hopkins University
susan@cs.jhu.edu

Brent Waters[†]
UT Austin
bwaters@cs.utexas.edu

January 18, 2018

Abstract

In this work we construct efficient aggregate signatures from the RSA assumption in the synchronized setting. In this setting, the signing algorithm takes as input a (time) period t as well the secret key and message. A signer should sign at most once for each t. A set of signatures can be aggregated so long as they were all created for the same period t. Synchronized aggregate signatures are useful in systems where there is a natural reporting period such as log and sensor data, or for signatures embedded in a blockchain protocol where the creation of an additional block is a natural synchronization event.

We design a synchronized aggregate signature scheme that works for a bounded number of periods T that is given as a parameter to a global system setup. The big technical question is whether we can create solutions that will perform well with the large T values that we might use in practice. For instance, if one wanted signing keys to last up to ten years and be able to issue signatures every second, then we would need to support a period bound of upwards of 2^{28}.

We build our solution in stages where we start with an initial solution that establishes feasibility, but has an impractically large signing time where the number of exponentiations and prime searches grows linearly with T. We prove this scheme secure in the standard model under the RSA assumption with respect to honestly-generated keys. We then provide a tradeoff method where one can tradeoff the time to create signatures with the space required to store private keys. One point in the tradeoff is where each scales with $\sqrt{T}$.

Finally, we reach our main innovation which is a scheme where both the signing time and storage scale with $\lg T$ which allows for us to keep both computation and storage costs modest even for large values of T. Conveniently, our final scheme uses the same verification algorithm, and has the same distribution of public keys and signatures as the first scheme. Thus we are able to recycle the existing security proof for the new scheme.

We also show how to extend our results to the identity-based setting in the random oracle model, which can further reduce the overall cryptographic overhead. We conclude with a detailed evaluation of the signing time and storage requirements for various practical settings of the system parameters.

[*] Supported by the National Science Foundation CNS-1228443 and CNS-1414023, the Office of Naval Research N00014-15-1-2778, and a Microsoft Faculty Fellowship.

[†] Supported by NSF CNS-1414082, DARPA SafeWare, Microsoft Faculty Fellowship, and Packard Foundation Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense or the U.S. Government.

1 Introduction

Aggregate signatures, as introduced by Boneh, Gentry, Lynn and Shacham [13], allow a third party to compress an arbitrary group of signatures $(\sigma_1, \ldots, \sigma_n)$ that verify with respect to a corresponding collection of public key and message pairs $((pk_1, m_1), \ldots, (pk_n, m_n))$ and produce a short aggregated signature that verifies the same collection. There exist many applications where reducing the cryptographic overhead is desirable including BGP routing [13, 13, 28, 11], bundling software updates [1], sensor network data [1] and block chain protocols [2].

When exploring a primitive such as aggregate signatures, it is desirable to have multiple realizations under different cryptographic assumptions or constructs. This provides redundancy in the case that one of the assumptions proves to be false. Also different approaches often yield a menu of performance tradeoffs that one can select from in an application-dependent manner.

To date, the design of aggregate signature schemes has mostly been dominated by bilinear (or multilinear) map-based proposals [13, 14, 10, 19, 28, 7, 11, 30, 35, 1, 18, 23, 22]. Most proposals to aggregate outside of the bilinear setting have required signers to interact either by signing in a sequential chain [29, 32, 17, 27, 26, 15, 4] or otherwise cooperate interactively on signature creation or verification [8, 3]. Here we seek a solution that does not require bilinear maps or signer interaction. We are aware of two prior attempts [36, 20] to aggregate RSA-based signatures (without interaction), but to the best of our understanding, both schemes appear to lack basic correctness (that is, each user creates and signs with his own unique modulus, but then the signatures are aggregated and verified with respect to the same modulus).

In this work we construct efficient aggregate signatures from the RSA assumption in the *synchronized* setting of Gentry and Ramzan [19]. In the synchronized setting the signing algorithm will take as input a (time) period t as well the secret key and message. A signer should sign at most once for each t. A set of signatures can be aggregated so long as they were all created for the same period t. Synchronized aggregate signatures are useful in systems where there is a natural reporting period such as log or sensor data. Another example is for use in signatures embedded in a blockchain protocol where the creation of an additional block is a natural synchronization event. For instance, consider a blockchain protocol that records several signed transactions in each new block creation. These signed transactions could use a synchronized aggregate signature scheme with the block iteration as the period number. This would reduce the signature overhead from one per transaction to only one synchronized signature per block iteration.

Ahn, Green and Hohenberger [1] gave a synchronized aggregate signature scheme in bilinear groups from the (standard model) computational Diffie-Hellman assumption by adapting the Hohenberger-Waters [24] short signature scheme. Since Hohenberger and Waters in the same work also provided a similar scheme from the RSA assumption it is natural to wonder why that one could not be adapted as well. Unfortunately, this approach will not work as the HW RSA-based signature scheme requires the signer to have knowledge of $\phi(N)$ and thus the factorization of N. This trapdoor information cannot be securely dispensed among all signers that might work in $\mathbb{Z}_N^*$.

In this work we design a synchronized aggregate signature scheme that works for a bounded number of periods T that is given as a parameter to a global system setup. We believe that such a bound is acceptable in the synchronized setting where a reasonable estimate of it can be derived by first determining a fixed lifetime of keys in the system (e.g., 10 years) and dividing it by the expected frequency that periods will occur (e.g., every minute). The big question is whether we can create solutions that will perform well with the larger T values that we might use in practice. For instance, suppose that we wanted signing keys to last up to ten years and wanted to have the capability of signing on periods as short as a second. In this case we would need to be able to support a period bound of upwards of 2^{28}.

We will build our solution in stages where we start with an initial solution that establishes feasibility of synchronized aggregation in the RSA setting, but has an impractically large signing time where the number of exponentiations and prime searches grows linearly with T. We prove this scheme secure in the standard model under the RSA assumption. We then provide a basic tradeoff that allows one to tradeoff the time to create signatures with the space required to store private keys. One point in the tradeoff is where each scales with $\sqrt{T}$.

We reach our main innovation which is a scheme where both the signing time and storage scale with $\lg(T)$ which allows for us to keep both computation and storage costs modest even for large values of T. Conveniently, our final scheme uses the same verification algorithm, and has the same distribution of public keys and signatures as the first scheme. Thus we are able to recycle the existing security proof for the new scheme.

We continue our exploration of using RSA in the synchronized aggregate setting by demonstrating how to extend our results to be identity-based. Since identity strings are typically much shorter than public keys, this setting can help achieve better overall reduction of cryptographic overhead. Our solution is secure under the standard RSA assumption in the random oracle model.

Finally, we provide a detailed performance evaluation of the various schemes proposed from both a signing time and private key storage perspective, concluding that the $\lg(T)$ construction is relatively practical for realistic settings of the system parameters and far exceeds the performance of the others for most settings.

Overview of the Schemes In our schemes, messages will be of length L bits which will be broken up into k chunks of ℓ bits each. In our initial scheme a global system setup will first choose an RSA modulus $N = p \cdot q$ where we let g be a generator of the quadratic residues of $\mathbb{Z}_N^*$. Next it picks a key K that is used to define a hash function $H_K(t) = e_t$ that maps a period $t \in [1, T]$ to a prime value e_t. We will defer the details of how this function works to the main body. Finally, the setup computes $E = \prod_{j=1}^{T} e_j \mod \phi(N)$ and $Y = g^E \mod N$ and publishes the public parameters as $\mathsf{pp} = (T, N, g, Y, K)$.

Key generation is performed by choosing random $u_0, u_1, \ldots, u_k$ in $[1, N]$ and setting the secret key as $sk = (u_0, u_1, \ldots, u_k)$ and the public key $pk = (U_0, U_1, \ldots, U_k)$ where $U_j = Y^{u_j} = g^{u_j \prod_{i \in T} e_i}$, for $j = 0$ to k. To sign a message first compute all the primes $e_i \leftarrow H_K(i)$ for $i \neq t$ and then output $\sigma = \left(g^{u_0} \prod_{j=1}^{k} g^{u_j \cdot m_j}\right)^{\prod_{i \in T \setminus \{t\}} e_i} = \left(U_0 \prod_{j=1}^{k} U_j^{m_j}\right)^{1/e_t} \pmod N$. Verification is performed by testing if $\sigma^{e_t} \overset{?}{=} U_0 \prod_{i=1}^{k} U_i^{m_i}$. Aggregation is done by simply multiplying individual signatures together (mod N) and testing against the product of the individual verification tests. We remark that our group hash function falls into a more general group hash framework proposed by Hofheinz, Jager and Kiltz [21]. In Section 4, we discuss potential future improvements by incorporating their framework.

We give a proof of security under the RSA assumption. Our proof is standard model with respect to honestly-generated keys and uses techniques from [24] for embedding an RSA challenge into the function H_K. The choice of k provides a tradeoff between the secret key storage size which grows linearly with k to the tightness in the reduction which has a loss factor of $2^\ell = 2^{L/k}$.

Taking a step back, our signature scheme involves reconstructing e_t-th roots of a public key and then manipulating these according to the message. Here the secret key simply holds a group element that is root of *all* the e_i values. The underlying structure is reminiscent of earlier RSA-based accumulator schemes (e.g., [9, 6]). The problem, however, is that building up this root from the secret key is quite costly and requires $T - 1$ exponentiations and calls to $H_K(\cdot)$ which are roughly equivalent to prime searches. Returning to our example of $T = 2^{28}$, our measured cost of signing one message was more than one day on a common processor. Clearly, we must do better.

We next show how to obtain a basic tradeoff between the time to sign and the size of the private key storage. Very roughly the time to sign will scale linearly with a parameter a and the storage with a parameter b with the constraint that $a \cdot b = T$. Thus we can explore tradeoffs such as setting $a = T, b = 1$ which corresponds to the scheme above, go the opposite direction and set $a = 1, b = T$ to achieve fast signing at the expense of large storage, or try to balance these by choosing $a = b = \sqrt{T}$.

The main technical idea is for the key generation algorithm to organize T into b "windows" each of size a. (We will assume a divides T evenly for ease of exposition.) Each window will be connected with a group element that has g raised to the exponents associated with every period except for a window of a of them. Thus to sign we need to do $a - 1$ exponentiations and prime searches and our private keys roughly grow as b group elements.

While this simple tradeoff technique provides more flexibility, there is still a significant gap from the performance numbers we would like to achieve. Let's return again to our $T = 2^{28}$ example. In setting $a = 1$, we would get very fast signing (a few tens of milliseconds), but with very huge keys of 64GB. On the other hand, if we aimed for the $\sqrt{T}$ tradeoff we would end up with 4MB private keys and roughly 9 seconds per signature. This achieves greater balance, but is still impractical.

This finally moves us to our last solution. Here we wish to find a more intricate way of handling the key storage that allows us to sign efficiently, but without a significant storage penalty. To do this we design a key storage mechanism that has about $2\lg(T)$ group elements and requires $\lg(T)$ exponentiations per signing. Returning to our example of $T = 2^{28}$, we can now achieve the much more practical 16KB private key storage with 58 milliseconds per signature.

To achieve this, we leverage the fact that the synchronized signatures are performed in sequence over the total number of periods. The goal is to maintain a data structure which (1) is small, (2) is ready to quickly produce a signature for the next period and (3) can perform a small amount of work to update it for future periods. To this end we organize a data structure according to a `levels` parameter where $T = 2^{\texttt{levels}+1} - 2$. In addition, a current `index` value is associated with the structure that indicates how many periods have passed so far. At level i at any time there will be one or two tuples which include a group element which is g raised to all exponents corresponding to periods except those with indices anywhere from 2^i to 2^{i-1}. During each signature the signing algorithm will grab an element from level 1 and use it to sign as well as perform a little bit of work on each level to close the window of exponents further. We defer the details of how this is achieved to Section 6. We remark that this approach is conceptually similar to the pebbling optimization used by Itkis and Reyzin [25] to realize efficient forward-secure signatures.

Organization and Summary of the Results In Section 2, we provide the specifications and security definitions. Section 3 covers the algebraic setting, assumptions and related lemmas. Section 4 gives the base construction as well as its proof of security in the standard model under the RSA assumption. Section 5 describes changes to the key generation and signing algorithms that can achieve a tradeoff in private key size versus signing time; one point achieves a balance of $\sqrt{T}$ for both. Section 6 provides a deeper innovation on how change key generation and signing to scale with $\lg(T)$. Recall that the distribution of the public keys and signatures in all of these schemes are the same as are the verification algorithms and thus the security proof in Section 4 suffices for all. We then show how to extend these results to the identity-based setting in Section 7 and provide the new definition and security proof required in Appendix A. Finally, we conclude with a detailed time and space performance analysis of these constructions in Section 8 showing that the $\lg(T)$ constructions can be practical even for very large bounds on T.

2 Scheme Specifications and Definitions of Security

In a basic aggregate signature scheme [13], anyone given n signatures on n messages from n users can aggregate all these signatures into a single short signature. This aggregate signature (together with the n public keys and n messages) can be publicly verified to convince anyone that user i authenticated message i for $i = 1$ to n. This is also true for synchronized aggregate signatures except that we assume all signers have a synchronized clock and the following restrictions apply:

1. A signer can issue at most one signature per period and keeps state to ensure this.

2. Only signatures created during the same period can be aggregated.

Gentry and Ramzan [19] were the first to consider this "synchronized" setting in the context of aggregate signatures. In their model, they assumed that signatures were issued using a special tag (which could not be re-used) and only signatures with the same tag could be aggregated. Ahn, Green and Hohenberger [1] formalized this synchronization as a time period, assuming all signers have access to the same clock. [1] In this work, we include a bound T on the periods.

Definition 2.1 (Synchronized Aggregate Signatures [19, 1]) *A synchronized aggregate signature scheme for a bounded number of periods and message space $\mathcal{M}(\cdot)$ is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$ *such that*

Setup$(1^\lambda, 1^T)$ *: On input the security parameter λ and the period bound T, the setup algorithm outputs public parameters* pp.

KeyGen(pp) *: On input the public parameters* pp*, the key generation algorithm outputs a keypair (pk, sk).*

Sign(pp, sk, M, t) *: On input the public parameters* pp*, the signing algorithm takes in a secret key sk, a message $M \in \mathcal{M}(\lambda)$, the current period $t \leq T$, and produces a signature σ.*

Verify$(\mathsf{pp}, pk, M, t, \sigma)$ *: On input the public parameters* pp*, the verification algorithm takes in a public key pk, a message M, a period t and a purported signature σ, and returns 1 if and only if the signature is valid and $t \leq T$, and 0 otherwise.*

Aggregate$(\mathsf{pp}, t, (pk_1, M_1, \sigma_1), \ldots, (pk_n, M_n, \sigma_n))$ *: On input the public parameters* pp*, a period t, a sequence of public keys $(pk_1, \ldots, pk_n)$, messages $(M_1, \ldots, M_n)$, and purported signatures $(\sigma_1, \ldots, \sigma_n)$ for period $t \leq T$, it outputs an aggregate signature σ_{agg} or error message $\perp$.*

[1] In this work, as in the case of [1], if the signers' clocks become out of sync with each other, this will lead to inefficiencies in the system, as it will not be possible to aggregate some signatures, but this will not open up security issues. As in [19, 1], there is a security issue if a tag or time period is reused by the signer, so an adversary's ability to move a user's clock backward could lead to forgeries for that signer.

AggVerify$(\mathsf{pp}, t, (pk_1, \ldots, pk_n), (M_1, \ldots, M_n), \sigma_{agg})$ *: On input the public parameters* pp*, a period* t*, a sequence of public keys* $(pk_1, \ldots, pk_n)$ *and messages* $(M_1, \ldots, M_n)$*, and a purported aggregate signature* σ_{agg}*, the aggregate-verification algorithm outputs 1 if and only if* σ_{agg} *is a valid aggregate signature and* $t \leq T$*, and 0 otherwise.*

Efficiency *We require that the setup algorithm run in time polynomial in its inputs and all other algorithms run in time polynomial in* λ, T*.*

Correctness *Let* $\mathsf{poly}(x)$ *denote the set of polynomials in* x*. In addition to the standard correctness properties of the basic signature scheme, for a synchronized aggregation scheme, the correctness requirements on* $\mathsf{Aggregate}$ *and* $\mathsf{AggVerify}$ *stipulate that for all* $\lambda \in \mathbb{N}$*,* $T \in \mathsf{poly}(\lambda), n \in \mathsf{poly}(\lambda)$*,* $\mathsf{pp} \in \mathsf{Setup}(1^\lambda, 1^T)$*,* $(pk_1, sk_1), \ldots, (pk_n, sk_n) \in \mathsf{KeyGen}(\mathsf{pp})$*,* $1 \leq t \leq T$*,* $M_i \in \mathcal{M}(\lambda)$*,* $\sigma_i \in \mathsf{Sign}(\mathsf{pp}, sk_i, M_i, t)$ *and* $\sigma_{agg} \in \mathsf{Aggregate}(\mathsf{pp}, t, (pk_1, M_1, \sigma_1), \ldots, (pk_n, M_n, \sigma_n))$*, it holds that*

$$\mathsf{AggVerify}(\mathsf{pp}, t, (pk_1, \ldots, pk_n), (M_1, \ldots, M_n), \sigma_{agg}) = 1.$$

Unforgeability The definition uses the following game between a challenger and an adversary $\mathcal{A}$ for a given scheme $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$, security parameter λ, and message space $\mathcal{M}(\lambda)$:

Setup: The adversary sends $1^T, 1^n$ to the challenger, who then runs $\mathsf{Setup}(1^\lambda, 1^T)$ to obtain the public parameters pp.[2] Then the challenger runs $\mathsf{KeyGen}(\mathsf{pp})$ a total of n times to obtain the key pairs $(pk_1, sk_1), \ldots, (pk_n, sk_n)$. The adversary is sent $(\mathsf{pp}, pk_1, (pk_2, sk_2), \ldots, (pk_n, sk_n))$.

Queries: For each period t starting with 1 and incrementing up to T, the adversary can request one signature on a message of its choice in $\mathcal{M}$ under sk_1, or it can choose to skip that period. The challenger responds to a query for M_i during period $t_i \in [1, T]$ as $\mathsf{Sign}(\mathsf{pp}, sk_1, M_i, t_i)$.

Output: Let γ be a function mapping integers to $[1, n]$. Eventually, the adversary outputs a tuple $(t, (pk_{\gamma(1)}, \ldots, pk_{\gamma(k)}), (M'_1, \ldots, M'_k), \sigma_{agg})$ and wins the game if:

1. $1 \leq t \leq T$; and
2. there exists an $z^* \in [1, k]$ such that $\gamma(z^*) = 1$; and
3. all $M'_i \in \mathcal{M}$; and
4. M'_{z^*} is not in the set of messages $\mathcal{A}$ queried during the Queries phase[3]; and
5. $\mathsf{AggVerify}(\mathsf{pp}, t, (pk_{\gamma(1)}, \ldots, pk_{\gamma(k)}), (M'_1, \ldots, M'_k), \sigma_{agg}) = 1$, where $1 \leq k \leq n$.

We define $\mathsf{SigAdv}_{\mathcal{A},\Pi,\mathcal{M}}(\lambda)$ to be the probability that the adversary $\mathcal{A}$ wins in the above game with scheme Π for message space $\mathcal{M}$ and security parameter λ taken over the coin tosses made by $\mathcal{A}$ and the challenger.

[2] For any adversary $\mathcal{A}$ that runs in time polynomial in λ will be restricted (by its own running time) to giving T values out that are polynomial in λ.

[3] As observed by [1], one can relax this unforgeability condition to allow the forgery message, M'_{z^*}, to have been previously queried to the signing oracle provided that it was not done during the same period used in the forgery. This "stronger" notion can be achieved by any scheme satisfying the above unforgeability definition by having the signer incorporate the period into each message.

Definition 2.2 (Unforgeability) *A synchronized aggregate signature scheme Π for message space $\mathcal{M}$ is existentially unforgeable under an adaptive chosen message attack if for all sufficiently large $\lambda \in \mathbb{N}$ and all probabilistic polynomial-time in λ adversaries $\mathcal{A}$, there exists a negligible function* negl*, such that* $\mathsf{SigAdv}_{\mathcal{A},\Pi,\mathcal{M}}(\lambda) \leq \mathsf{negl}(\lambda)$.

Discussion In our definition, we require that the Setup algorithm is honestly executed, so in practice this could be run by a trusted party or realized via a specialized multiparty protocol (see Section 4 for more). We also require that the non-challenge public keys be chosen honestly instead of adversarially. Our later proof requires that the challenger has knowledge of the secret keys corresponding to the non-challenge public keys. This can be realized by working in the Registered Key Model [5] or adding an appropriate NIZK to the user's public key.

3 Number Theoretic Assumptions and Related Lemmas

There are many variants of the RSA assumption [34]. Here we use a variant involving safe primes. A *safe prime* is a prime number of the form $2p + 1$, where p is also a prime.

Assumption 3.1 (RSA) *Let λ be the security parameter. Let integer N be the product of two λ-bit, distinct safe primes primes p, q where $p = 2p' + 1$ and $q = 2q' + 1$. Let e be a randomly chosen prime between 2^λ and $2^{\lambda+1} - 1$. Let QR_N be the group of quadratic residues in $\mathbb{Z}_N^*$ of order $p'q'$. Given (N, e) and a random $h \in \mathrm{QR}_N$, it is hard to compute x such that $x^e \equiv h \mod N$.*

We note that a randomly chosen element in $\mathbb{Z}_N^*$ would be a quadratic residue 1/4-th of the time, so the restriction to $h \in \mathrm{QR}_N$ is for convenience and could be relaxed.

In our schemes, we will refer to and require a primality test. For our purposes, it will be sufficient to use the efficient Miller-Rabin test [31, 33]. We will also make use of the following lemmas:

Lemma 3.2 (Cramer-Shoup [16]) *Given $x, y \in \mathbb{Z}_n$ together with $a, b \in \mathbb{Z}$ such that $x^a = y^b$ and $\gcd(a, b) = 1$, there is an efficient algorithm for computing $z \in \mathbb{Z}_n$ such that $z^a = y$.*

Theorem 3.3 (Prime Number Theorem) *Define $\pi(x)$ as the number of primes $\leq x$. For $x > 1$,*

$$\frac{7}{8} \cdot \frac{x}{\ln x} < \pi(x) < \frac{9}{8} \cdot \frac{x}{\ln x}.$$

4 A Base Scheme for Aggregation from RSA

We begin with a base scheme that assumes a trusted global setup and works in the registered key model, where every signer needs to show their key pair to an authority that certifies their public key. The global setup of our scheme will take as input a security parameter λ and the maximum number of periods T. The message space $\mathcal{M}$ will be $\{0, 1\}^L$ where L is some polynomial function of λ. (One can handle messages of arbitrary length by first applying a collision-resistant hash.)

In addition, associated with the scheme will be a "message chunking alphabet" where we break each L-bit message into k chunks each of ℓ bits where $k \cdot \ell = L$ with the restriction that $\ell \leq \lambda$ and thus $2^\ell \leq 2^\lambda$. As we will see the choice of ℓ will effect both the tightness of the security reduction as well as the size of the signatures.[4] We make use of a variant of the hash function in [24] to map integers to primes of an appropriate size.

Setup$(1^\lambda, 1^T)$ The setup algorithm chooses an integer $N = pq$ as the product of two safe primes where $p - 1 = 2p'$ and $q - 1 = 2q'$, such that $2^\lambda < \phi(N) < 2^{\lambda+1}$. Let QR_N denote the group of quadratic residues of order $p'q'$ with generator g.

Next, it sets up a hash function $H : [1, T] \to \{0, 1\}^{\lambda+1}$ where H will take as input a period $t \in [1, T]$ and output a prime between 2^λ and $2^{\lambda+1} - 1$. It begins by randomly choosing a K' for the PRF function $F : [1, T] \times [1, \lambda^2] \to \{0, 1\}^\lambda$, a random $c \in \{0, 1\}^\lambda$ as well as an arbitrary prime e_{default} between between 2^λ and $2^{\lambda+1} - 1$. We let $K = (K', c, e_{\text{default}})$.

We define how to compute $H_K(t)$. For each $i = 1$ to $\lambda \cdot (\lambda^2 + \lambda)$, let $y_i = c \oplus F_K(t, i)$. If $2^\lambda + y_i$ is prime return it. Else increment i and repeat. If no such $i \leq \lambda \cdot (\lambda^2 + \lambda)$ exists, return e_{default}.[5] We note that this computation returns the smallest i such that $2^\lambda + y_i$ is a prime. Notationally, for $t \in [1, T]$ we will let $e_t = H_K(t)$.

The algorithm concludes by computing $E = \prod_{j=1}^{T} e_j \mod \phi(N)$ and $Y = g^E \mod N$.

It publishes the public parameters as $\mathsf{pp} = (T, N, g, Y, K)$.

KeyGen(pp) The algorithm retrieves Y from the pp. It chooses random integers $u_0, u_1, \ldots, u_k$ in $[1, N]$. It sets the secret key as $sk = (u_0, u_1, \ldots, u_k)$ and the public key $pk = (U_0, U_1, \ldots, U_k)$ where

$$U_j = Y^{u_j} = g^{u_j \prod_{i \in T} e_i}, \text{ for } j = 0 \text{ to } k.$$

Sign(pp, sk, M, t) The signing algorithm takes as input a time period $1 \leq t \leq T$ and an $L = (\ell k)$-bit message $M = m_1|m_2|\ldots|m_k$, where each m_i contains ℓ-bits and these are concatenated together to form M. It computes the primes $(e_1, \ldots, e_{t-1}, e_{t+1}, \ldots, e_T)$ from pp and then outputs

$$\sigma = \big(g^{u_0} \prod_{j=1}^{k} g^{u_j \cdot m_j}\big)^{\prod_{i \in T \setminus \{t\}} e_i} = \big(U_0 \prod_{j=1}^{k} U_j^{m_j}\big)^{1/e_t} \pmod{N}.$$

Verify$(\mathsf{pp}, pk, M, t, \sigma)$ Let $M = m_1|m_2|\ldots|m_k$. The algorithm computes the prime e_t from pp. Output 1 if $1 \leq t \leq T$ and

$$\sigma^{e_t} \stackrel{?}{=} U_0 \prod_{i=1}^{k} U_i^{m_i} \pmod{N}$$

or 0 otherwise.

Aggregate$(\mathsf{pp}, t, (pk_1, M_1, \sigma_1), \ldots, (pk_n, M_n, \sigma_n))$ An aggregate signature on signatures from the same time period $1 \leq t \leq T$ is computed as $\sigma_{agg} = \prod_{j=1}^{n} \sigma_j \pmod{N}$.

[4] In practice, one might use a collision-resistant hash function to map arbitrarily long messages into $L = 256$ bits and then set $\ell = 32$ and $k = 8$. We discuss the efficiency implications of these choices in Section 8.

[5] We expect this default case to be exercised only with negligible probability, but define it so that the function $H_K(t)$ is guaranteed to terminate in a bounded amount of time.

AggVerify$(\mathsf{pp}, t, (pk_1, \ldots, pk_n), (M_1, \ldots, M_n), \sigma_{agg})$ Let $pk_j = (U_{j,0}, U_{j,1}, \ldots, U_{j,k})$ and $M_j = m_{j,1}|m_{j,2}|\ldots|m_{j,k}$. The algorithm computes the prime e_t from pp. Output 1 if $1 \leq t \leq T$, each public key is unique (i.e., $\forall i \neq j \in [1, n]$, $pk_i \neq pk_j$) and

$$\sigma_{agg}^{e_t} \stackrel{?}{=} \prod_{j=1}^{n} (U_{j,0} \prod_{i=1}^{k} U_{j,i}^{m_{j,i}}) \pmod N$$

or 0 otherwise.

Discussion Observe that the above group hash function we employ falls into a more general group hash framework proposed by Hofheinz, Jager and Kiltz [21] that uses programmable hash functions. One might use their general framework to explore further concrete efficiency tradeoffs, such as letting the group hash function be more complex and letting the hash function output the product of multiple smaller primes. Our concrete analysis, however, will focus on the core scheme above along with tradeoffs in key storage and signing time that we explore later. We leave open the interesting question of what other tradeoffs can be realized via [21], keeping in mind that some of those instantiations add per signer randomness, which makes aggregation challenging.

Recall from Section 2 that the **Setup** algorithm must be executed honestly. It seems promising that, for this scheme, this might be realized efficiently using a specialized multiparty computation protocol, such as an adaptation of one due to Boneh and Franklin [12] for efficiently allowing a group of parties to generate an RSA modulus, where each party learns N, but no party learns the factorization of N.

4.1 Proof of Security

Theorem 4.1 *If the RSA assumption (Assumption 3.1) holds and F is a secure pseudorandom function, then the above synchronized aggregate signature construction is existentially unforgeable under an adaptive chosen message attack.*

Proof. The reduction algorithm receives an RSA challenge (N, e^*, h) and needs to use the attacker to compute $h^{1/e^*} \mod N$. Define a "conforming" attacker as one that will always make a signing query on the period t^* that it forges on. We can assume our attacker is conforming without loss of generality because if there exists an attacker that breaks the scheme, there exits one that breaks it and queries for a signature on period t^* by simply adding a signature query on a random message at that period. Our proof will assume a conforming attacker.

Next, we define a sequence of games.

Game 1: (Security Game) This game is defined to be the same as the security game of the scheme.

Game 2: (Guessing the forgery period and part of its queried message) The same as Game 1, except the game guesses the period the attacker will forge on and a part of the message queried for a signature during the period that will be different from the forgery message, and the adversary only wins if these guesses were correct. Formally, the game chooses random $t' \in [1, T]$, $\alpha \in [1, k]$ and $\beta \in \{0, 1\}^{\ell}$. An adversary wins this game iff: (1) it would have won in Game 1 with a forgery on period t^* for some message $M^* = m_1^*|m_2^*|\ldots|m_k^*$ with some message $M = m_1|m_2|\ldots|m_k$ queried to the signing oracle on period t^*, (2) $t' = t^*$, (3) $\beta = m_\alpha$ and (4) $m_\alpha \neq m_\alpha^*$.

Game 3: (H_K does not default) The attacker wins only if it meets all the conditions to win in Game 2 and $H_K(t^*) \neq e_{\text{default}}$ (that is, the default condition of the hash is not triggered on the forgery message or otherwise equal to the default prime.)

Game 4: (H_K does not collide) The attacker wins only if it meets all the conditions to win in Game 3 and $H_K(t^*) \neq H_K(t)$ for all $t \in [1, T]$ where $t \neq t^*$.

Game 5: (Guess resolving i^* for H_K) The game chooses a random $i^* \in [1, \lambda^3 + \lambda^2]$. Attacker wins only if it meets all the conditions of Game 4 and i^* was the "resolving" index in $H_K(t^*)$; that is, i^* was the smallest i such that $y_i = F_{K'}(t^*, i) \oplus c$ and $(2^\lambda + y_i)$ was a prime.

Game 6: (Programming H_K with random value) The same as Game 5, except that it chooses a random $y' \in \{0,1\}^\lambda$ and set $c = y' \oplus F_{K'}(t^*, i^*)$.

Game 7: (Programming H_K with e^*) The same as Game 6, except choose e^* as a random prime in the range $[2^\lambda, 2^{\lambda+1} - 1]$ and let y' be the λ least significant bits of e^*; that is, drop the leading 1. As before, set $c = y' \oplus F_{K'}(t^*, i^*)$.

We now establish a series of claims that show that if an adversary is successful against the real security game (Game 1) then it will be successful against in Game 7 as well. We will then shortly describe a simulator that can use any adversary successful in Game 7 to solve the RSA challenge.

Define $\mathbf{Adv}_{\mathcal{A}}[\text{Game } x]$ as the advantage of an adversary $\mathcal{A}$ in Game x.

Claim 4.2

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 2}] \geq \frac{\mathbf{Adv}_{\mathcal{A}}[\text{Game 1}]}{T \cdot k \cdot 2^\ell}.$$

Proof. Since there is no change to the adversary's view of the game, the probability of the adversary winning in Game 2 is the same as Game 1 times the probability of the game's guesses being correct. There is a $1/T$ probability of guessing the forging period, at least a $1/k$ probability of guessing a message chunk in the signing query that will be different in the forgery (there may be more than one), and a 2^ℓ probability of guessing that chunk's value in the queried message. We note that this gives a polynomial-time reduction for whenever ℓ is polylogarithmic in λ. Recall that any adversary that is polynomial time in λ must give out a 1^T that is polynomially bounded in λ. □

Claim 4.3 *If F is a secure pseudorandom function and $\lambda \geq 4$, then*

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 3}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 2}] - \mathsf{negl}(\lambda).$$

Proof. We here need to understand the probability that $H_K(t^*) = e_{\text{default}}$. Using the Prime Number Theorem, we can bound the number of primes in the range $[2^\lambda, 2^{\lambda+1} - 1]$ as follows. Plugging into the formula in Lemma 3.3, we have that the number of primes less than $2^{\lambda+1} - 1$ is at least $\frac{7}{8} \cdot \frac{2^{\lambda+1}}{(\lambda+1)}$ (the value $2^{\lambda+1}$ is not prime, since it is a power of two, for any $\lambda \geq 1$) and the number of primes less than 2^λ is at most $\frac{9}{8} \cdot \frac{2^\lambda}{\lambda}$. Thus, the total number of primes in our range of interest is at least

$$\frac{7}{8} \cdot \frac{2^{\lambda+1}}{(\lambda+1)} - \frac{9}{8} \cdot \frac{2^\lambda}{\lambda} = \frac{7 \cdot \lambda \cdot 2^{\lambda+1} - 9 \cdot (\lambda+1) \cdot 2^\lambda}{8(\lambda+1)\lambda} \qquad = \frac{14 \cdot \lambda \cdot 2^\lambda - 9 \cdot (\lambda+1) \cdot 2^\lambda}{8(\lambda+1)\lambda} \tag{1}$$

$$= \frac{5 \cdot \lambda \cdot 2^\lambda - 9 \cdot 2^\lambda}{8(\lambda+1)\lambda} \qquad = \frac{(5\lambda - 9) \cdot 2^\lambda}{8(\lambda^2+\lambda)} \tag{2}$$

$$> \frac{2^\lambda}{\lambda^2+\lambda}, \text{ for all } \lambda \geq 4. \tag{3}$$

Let R be a random function that outputs a value in the range $[2^\lambda, 2^{\lambda+1}]$. Then the probability that R outputs a prime is at least:

$$\frac{2^\lambda/(\lambda^2+\lambda)}{2^{\lambda+1} - 2^\lambda} = \frac{2^\lambda}{2^\lambda(\lambda^2+1)} = \frac{1}{\lambda^2+\lambda} \tag{4}$$

The probability that R fails to output a prime after $\lambda(\lambda^2+\lambda)$ tries is as follows. We again use the fact that $2^{\lambda+1}$ is not a prime. Recall Chernoff's bound for any $\epsilon \geq 0$, we have $\Pr[X \leq (1-\epsilon)\mu] \leq e^{-\frac{\epsilon^2\mu}{2}}$. Here when X is the number of primes output by R in $\lambda(\lambda^2+\lambda)$ trials, $\epsilon = 1$ and $\mu = \sum^{\lambda(\lambda^2+\lambda)} \Pr[R \text{ fails to output a prime in one trial}]$, we have that

$$\Pr[R \text{ fails to output a prime in } \lambda^3 + \lambda^2) \text{ trials}] = \Pr[X \leq 0] \leq e^{-\frac{\mu}{2}} \leq e^{-\frac{\lambda(\lambda^2+\lambda)\cdot\frac{1}{\lambda^2+\lambda}}{2}} = e^{-\lambda/2} \tag{5}$$

The PRF we employ to sample from this range cannot non-negligibly differ from R in its probability of selecting primes or this provides for a distinguishing attack on the PRF. Thus, the probability that $H_K(t^*) = e_{\text{default}}$ is the probability that the PRF chose the same prime as the setup algorithm, which is negligible at 1 in the number of primes in that range ($> 2^\lambda/(\lambda^2+\lambda)$), plus the probability that H_K triggers the default condition by failing to output a prime, which we also argued was negligibly close to the negligible probability of R doing the same. □

Claim 4.4 *If F is a secure pseudorandom function and $T \in \mathsf{poly}(\lambda)$, then*

$$\mathbf{Adv}_\mathcal{A}[\text{Game 4}] = \mathbf{Adv}_\mathcal{A}[\text{Game 3}] - \mathsf{negl}(\lambda).$$

Proof. These games differ only in the event that $H_K(t^*) = H_K(t)$ for some $t \in [1, T]$ where $t \neq t^*$. Let R be a random function that outputs a value in the range $[2^\lambda, 2^{\lambda+1}]$. Suppose H_K uses R instead of the PRF. Then the probability of a collision for a single t is one in the number of primes in $[2^\lambda, 2^{\lambda+1}]$ or at most $1/\frac{2^\lambda}{\lambda^2+\lambda} = \frac{\lambda^2+\lambda}{2^\lambda}$, which is negligible. So the probability of a collision for any $t \in [1, T]$ (recall that T is polynomial in λ) is $T \cdot \frac{\lambda^2+\lambda}{2^\lambda} = \frac{\mathsf{poly}(\lambda)(\lambda^2+\lambda)}{2^\lambda} = \frac{\mathsf{poly}(\lambda)}{2^\lambda} = \mathsf{negl}(\lambda)$. When we replace R with the PRF, the probability of a collision cannot non-negligibly differ or this provides a distinguishing attack on the PRF. □

Claim 4.5

$$\mathbf{Adv}_\mathcal{A}[\text{Game 5}] = \frac{\mathbf{Adv}_\mathcal{A}[\text{Game 4}]}{\lambda^3 + \lambda^2}.$$

Proof. The attacker's view in these games is identical. The only difference is whether the game correctly guesses the resolving index i^* for $H_K(t^*)$. Since $i^* \in [1, \lambda^3+\lambda^2]$, the game has a $1/(\lambda^3+\lambda^2)$ chance of guessing this correctly. □

Claim 4.6

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 6}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 5}].$$

Proof. In Game 5, c is chosen randomly in $\{0,1\}^\lambda$. In Game 6, c is set by randomly selecting $y' \in \{0,1\}^\lambda$ and setting $c = y' \oplus F_{K'}(t^*, i^*)$, where t^* is the period on which the attacker will attack and i^* is the resolving index for this value. Since y' is chosen randomly and independently of $F_{K'}(t^*, i^*)$, the resulting c will be from the same distribution as Game 5. □

Claim 4.7

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 7}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 6}].$$

Proof. An adversary's advantage in these games is the same. In Game 6, the attacker could only win if $2^\lambda + y'$ was a prime, and thus the distributions are the same. □

4.1.1 Main Reduction

We now show that if there exists a polynomial-time (in λ) attacker that has advantage $\epsilon = \epsilon(\lambda)$ in Game 7, then there exists a polynomial-time (in λ) attacker for the RSA problem in Assumption 3.1 with advantage ϵ.

On input an RSA challenge (N, e^*, h), the reduction algorithm proceeds as follows:

Setup.

1. Obtain $1^T, 1^n$ from the aggregate signature adversary $\mathcal{A}$.
2. Make random guesses of $t^* \in [1,T], \alpha \in [1,k], \beta \in \{0,1\}^\ell, i^* \in [1, \lambda^3+\lambda^2]$.
3. Choose a random PRF key K'. Let y' be the λ least significant bits of the RSA input e^* (note that this is a prime randomly chosen from the appropriate range by the RSA challenger) and set $c = y' \oplus F_{K'}(t^*, i^*)$. Choose a random prime $e_{\text{default}} \in [2^\lambda, 2^{\lambda+1}-1]$. Set $K = (K', c, e_{\text{default}})$. Thus, note that by construction when i^* is the resolving index for t^*,
$$e_{t^*} = H_K(t^*) = 2^\lambda + (c \oplus F_{K'}(t^*, i^*)) = 2^\lambda + y' = e^*.$$
4. Choose a random $g \in \text{QR}_N$. Compute Y as before.
5. Set the $\mathsf{pp} = (T, N, g, Y, K)$.
6. Set up the "target" user's public key pk_1 as:
   (a) Choose random $u_0, u_1, \ldots, u_k \in [1, N]$.
   (b) Set $U_0 = (h^{-\beta})^{\prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_0}$. We note that the reduction algorithm can take the e_t root of U_0 so long as $t \neq t^*$.
   (c) For $j = 1$ to k such that $j \neq \alpha$, compute $U_j = Y^{u_j}$.
   (d) Set $U_\alpha = h^{\prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_\alpha}$.
7. Set $pk_1 = (U_0, U_1, \ldots, U_k)$. For $j = 2$ to n, $(pk_j, sk_j) = \mathsf{KeyGen}(\mathsf{pp})$.
8. Send to $\mathcal{A}$ the tuple $(\mathsf{pp}, pk_1, (pk_2, sk_2), \ldots, (pk_n, sk_n))$.

Queries. For each period $t = 1$ to T, the adversary can request one signature on a message of its choice in the message space under sk_1 or skip that period. Recall that the adversary must be conforming and thus will request some signature on the forgery period t^*. In our construction, signing during period t requires taking the e_t-th root of each U_j value. By construction, the reduction algorithm can do this so long as: (1) $t \neq t^*$ or (2) for t^*, when the α-th ℓ-bits of the message are the string β. If the reduction is ever asked a query it cannot answer, then it will abort. We note that this only occurs when the guesses of t^*, α, β are incorrect, which is consistent with the attacker not winning in Game 7 anyway. Formally, when asked to sign $M = m_1|m_2|\dots|m_k$ for period $t \neq t^*$, the reduction algorithm outputs:

$$\begin{aligned}
\sigma &= (h^{-\beta} \cdot h^{m_\alpha})^{\prod_{i\neq t^*, i\neq t}^{T} e_i} \cdot \big(g^{u_0} \prod_{j=1}^{k} g^{u_j m_j}\big)^{\prod_{i\in T\setminus\{t\}} e_i} && (6)\\
&= (h^{-\beta \prod_{i\neq t^*}^{T} e_i} \cdot Y^{u_0})^{1/e_t} \cdot (\prod_{j=1, j\neq\alpha}^{k} U_j^{m_j})^{1/e_t} \cdot (h^{\prod_{i\neq t^*}^{T} e_i} \cdot Y^{u_\alpha})^{m_\alpha/e_t} && (7)\\
&= (U_0 \prod_{j=1}^{k} U_j^{m_j})^{1/e_t} \mod N. && (8)
\end{aligned}$$

and when $t = t^*$ and $m_\alpha = \beta$, it outputs the signature:

$$\begin{aligned}
\sigma &= \big(g^{u_0} \prod_{j=1}^{k} g^{u_j m_j}\big)^{\prod_{i\in T\setminus\{t\}} e_i} && (9)\\
&= (1)^{\prod_{i\neq t^*, i\neq t}^{T} e_i} \cdot \big(g^{u_0} \prod_{j=1}^{k} g^{u_j m_j}\big)^{\prod_{i\in T\setminus\{t\}} e_i} && (10)\\
&= (h^{-\beta} \cdot h^{m_\alpha})^{\prod_{i\neq t^*, i\neq t}^{T} e_i} \cdot \big(g^{u_0} \prod_{j=1}^{k} g^{u_j m_j}\big)^{\prod_{i\in T\setminus\{t\}} e_i} && (11)\\
&= (h^{-\beta \prod_{i\neq t^*}^{T} e_i} \cdot Y^{u_0})^{1/e_t} \cdot (\prod_{j=1, j\neq\alpha}^{k} U_j^{m_j})^{1/e_t} \cdot (h^{\prod_{i\neq t^*}^{T} e_i} \cdot Y^{u_\alpha})^{m_\alpha/e_t} && (12)\\
&= (U_0 \prod_{j=1}^{k} U_j^{m_j})^{1/e_t} \mod N. && (13)
\end{aligned}$$

Output. Eventually $\mathcal{A}$ outputs a tuple $(t_f, (pk_{\gamma(1)}, \dots, pk_{\gamma(z)}), (M_1, \dots, M_z), \sigma_{agg})$. Since aggregation order does not matter here[6], we can w.l.o.g. assume that $\gamma(1) = 1$ (corresponding to the target key pk_1); we also drop γ from the subscript below. If the aggregate signature does not verify or if any of the reduction's guesses of t^*, i^*, α, β were incorrect, then abort. These abort conditions are all consistent with the adversary not winning Game 7. Let $E' = \prod_{i\in T\setminus\{t^*\}} e_i$. Otherwise we

[6] Consider any signature σ_{agg} that verifies on time period t for public keys $pk_1, \dots, pk_z$ and messages $M_1, \dots, M_z$. Our scheme has the property that σ_{agg} also verifies on any permutation of the indices applied to both the public key and message sequences.

have that:

$$
\begin{aligned}
\sigma_{agg}^{e^*} &= \prod_{j=1}^{n}(U_{j,0}\prod_{i=1}^{k}U_{j,i}^{m_{j,i}}) && (14)\\
&= (U_{1,0}\prod_{i=1}^{k}U_{1,i}^{m_{1,i}})\cdot\prod_{j=2}^{n}(U_{j,0}\prod_{i=1}^{k}U_{j,i}^{m_{j,i}}) && (15)\\
&= (h^{E'(\beta-m_\alpha)}\cdot Y^{u_0}\prod_{j=1}^{k}Y^{u_j m_j})\cdot\prod_{j=2}^{n}(U_{j,0}\prod_{i=1}^{k}U_{j,i}^{m_{j,i}}) && (16)
\end{aligned}
$$

Since the reduction can compute the e^*-th root of all values not in the h term, it can divide them out as:

$$
\begin{aligned}
\left(\frac{\sigma_{agg}}{\prod_{j=1}^{n}(g^{u_{j,0}}\prod_{i=1}^{k}g^{u_{j,i}m_{j,i}})^{E'}}\right)^{e^*} &= \frac{(h^{E'(\beta-m_\alpha)}\cdot Y^{u_0}\prod_{j=1}^{k}Y^{u_j m_j})\cdot\prod_{j=2}^{n}(U_{j,0}\prod_{i=1}^{k}U_{j,i}^{m_{j,i}})}{\prod_{j=1}^{n}(g^{u_{j,0}}\prod_{i=1}^{k}g^{u_{j,i}m_{j,i}})^{e^*\cdot E'}} && (17)\\
&= h^{E'(\beta-m_\alpha)}. && (18)
\end{aligned}
$$

Now, we have an equation of the form $x^a = y^b$, where $x = \frac{\sigma_{agg}}{\prod_{j=1}^{n}(g^{u_{j,0}}\prod_{i=1}^{k}g^{u_{j,i}m_{j,i}})^{E'}}$, $a = e^*$, $y = h$ and $b = E'(\beta - m_\alpha)$. Recall that the game would have already aborted if e^* was output for any period other than t^* and thus, $\gcd(e^*, E') = 1$. The game would also have aborted if $\beta = m_\alpha$. Finally since the $|\beta| = |m_\alpha| = \ell < \lambda$ and $e^* > 2^\lambda$, we can conclude that $\gcd(a, b) = 1$. This allows the reduction to apply Lemma 3.2 to efficiently compute $\hat{h} \in \mathbb{Z}_N$ such that $\hat{h}^{e^*} = h \mod N$. The reduction outputs this value as the RSA solution.

Analysis. We argue that the above reduction will succeed in outputting the RSA solution whenever the adversary wins in Game 7. The adversary's view in these scenarios differs only in the way that public key elements U_0 and U_α are chosen. We will first argue that the way they are chosen in Game 7 (and the actual scheme) is statistically close to choosing a random element in QR_N. Next, we argue that the (different) way they are chosen in the reduction above is also statistically close to choosing a random element in QR_N. It follows then that the public key in both Game 7 and the reduction are statistically close and thus cannot be distinguished by our polynomial-time adversary. Moreover, while the signatures are computed via a different method in Game 7 and the reduction, the signature the adversary sees is identical (and unique) given the public information known to the adversary, so there is no information the adversary can use to distinguish. For any given $U \in \mathrm{QR}_N$, prime $e \in [1, N]$, and $m < 2^\lambda$, the values U^{em} and $U^{1/e}$ are unique since each e_i is relatively prime to $\phi(N)$. It remains to support the arguments listed above.

First, recall how U_0, U_α are chosen in Game 7 (and the actual scheme). Here u_0, u_α are randomly chosen from $[1, N]$ and the public key elements are set as:

$$U_0 = Y^{u_0} = g^{u_0\prod_{i=1}^{T}e_i} \quad , \quad U_\alpha = Y^{u_\alpha} = g^{u_\alpha\prod_{i\in T}e_i}.$$

Observe that the group of QR_N has order $p'q'$. Thus $Y = g^{\prod_{i=1}^{T}e_i}$ is also a generator since all the e_i values are relatively prime to $p'q'$. Since Y is a generator, if we take Y^r for a random $r \in [1, \phi(N)]$ that has the same distribution as choosing a random element in QR_N. Now, the process of raising

Y^r for a random $r \in [1, N]$ is statistically close to the process of raising it to a random $r \in [1, \phi(N)]$. The reason is that $N = \phi(N) + p + q - 1$ where the difference $(p + q - 1)$ is negligible. Thus, we achieve our first argument.

Second, recall how U_0, U_α are chosen in the reduction. Here u_0, u_α are randomly chosen from $[1, N]$ and the public key elements are set as:

$$U_0' = (h^{-\beta})^{\prod_{i\neq t^*}^T e_i} \cdot Y^{u_0} = h^{-\beta \prod_{i\neq t^*}^T e_i} \cdot g^{u_0 \prod_{i=1}^T e_i} \quad , \quad U_\alpha' = h^{\prod_{i\neq t^*}^T e_i} \cdot Y^{u_\alpha}.$$

We previously argued that the Y^{u_0} and Y^{u_α} components are distributed statistically close to a random element in QR_N. We assume that $h \in \mathrm{QR}_N$; this will be true for a random element in $\mathbb{Z}_N^*$ with 1/4 probability. Each value has an h term that is in QR_N but not necessarily distributed randomly. However, once we multiply this value in the group by a (statistically close to) random element of the group, we have a product that is distributed statistically close to a random element in QR_N. Thus, we achieve our second argument.

Since the adversary cannot distinguish either distribution of public keys from a random distribution, then it cannot distinguish them from each other as well. Thus, whenever the adversary succeeds in Game 7, we can conclude it will also succeed in helping the reduction solve RSA.

□

5 Trading off Signing Time with Storage

In this section we show a basic tradeoff between the time to sign and the size of the private key storage. Very roughly the time to sign will scale linearly with a parameter a and the storage with a parameter b with the constraint that $a \cdot b = T$. Thus we can explore tradeoffs such as setting $a = T, b = 1$ as we saw in the last section or go the opposite direction and set $a = 1, b = T$ to achieve fast signing at the expense of large storage, or try to balance these by choosing $a = b = \sqrt{T}$.

Our system will use the same setup, verification and aggregation algorithms as in Section 4 and just replace the KeyGen and Sign algorithms. Moreover, the public keys output by the KeyGen algorithm and corresponding signatures output by the Sign algorithm will have an identical distribution to the original Section 4 scheme and thus not require a new security proof.

Let the public parameters output from Setup be $\mathsf{pp} = (T, N, g, Y, K)$ as before. Our KeyGeneration algorithm will organize T into b "windows" each of size a. (We will assume a divides T evenly for ease of exposition.) Then the private key will be setup to contain a sequence of values R_w which is g raised to all e_i except those in a sliding window of periods. To sign faster during time period t, select these partially computed values where t is in the window and complete its computation for signing by raising to all e_i in that window except e_t.

The new key generation and signing algorithms follow.

KeyGen$'(\mathsf{pp}, a)$ It obtains the primes $(e_1, \ldots, e_T)$ and sets $b = T/a$ (we assume it divides evenly for ease of exposition). Next it chooses random integers $u_0, u_1, \ldots, u_k$ in $[1, N]$ and computes $pk = (U_0, U_1, \ldots, U_k)$. For $w = 1$ to b, define Σ_w as the set of integers in $[1, T]$ other than those in the set $\{a(w-1)+1, a(w-1)+2, \ldots, a(w-1)+a\}$.

For $w = 1$ to b, it then computes:

$$R_w = g^{\prod_{i \in \Sigma_w} e_i}$$

where the e_i values are computed using K from pp. It sets the secret key as $sk = (\{R_w\}_{1\leq w \leq b}, \{u_i\}_{0 \leq i \leq k})$. The public key $pk = (U_0, U_1, \dots, U_k)$ is computed as $U_j = Y^{u_j} = g^{u_j \prod_{i \in T} e_i}$, for $j = 0$ to k as in Section 4.

Sign′(pp, sk, M, t) It computes the necessary subset of primes in $(e_1, \dots, e_T)$ using K in pp and then for period t, selects the window $w = \lceil t/a \rceil$. Let Σ'_w denote the set of periods in the window $\{a(w-1)+1, a(w-1)+2, \dots, a(w-1)+a\}_{1 \leq w \leq b}$. It outputs

$$\sigma = \big(R_w^{u_0} \prod_{j=1}^{k} R_w^{u_j \cdot m_j}\big)^{\prod_{i \in \Sigma'_w \setminus \{t\}} e_i} = \big(U_0 \prod_{j=1}^{k} U_j^{m_j}\big)^{1/e_t} \pmod{N}.$$

Analysis. Observe that the public keys and signatures are of the same form and distribution as those of the base system in Section 4 and thus the security of this tradeoff system follows.

6 Obtaining $O(\lg(T))$ Signing Time and Private Key Size

The previous section showed a basic tradeoff between signing time and private key size. However, it was limited in that the most "balanced" version required both time and storage to grow with the square root of the number of periods.

In this section we show how a more intricate key storage technique can give us much better results with a scheme where the number of exponentiations and prime searches is $\approx \lg(T)$ per signing operation and where we store $\approx \lg(T)$ elements of $\mathbb{Z}_N^*$ in the private key. Unlike the previous schemes where the private key remained static, our method here will require us to update the private key on each signing period. As a consequence a signer will be required to sign using each period in sequence.[7] Again, our new scheme will produce public keys and signatures with exactly the same distribution as the base scheme of Section 4. Therefore we will only need to describe and analyze the new method of key generation and storage and are not required to produce a new security proof. As mentioned earlier, this approach has conceptual roots in the pebbling optimization used by Itkis and Reyzin [25] to realize efficient forward-secure signatures.

We present our method by introducing two algorithms. The first algorithm $\texttt{StorageInit}(\mathsf{pp}, v)$ takes in the public parameters and an element $v \in \mathbb{Z}_N^*$ and outputs the initial key storage state $\texttt{store}$. The second algorithm $\texttt{StorageUpdate}(\texttt{store})$ takes in the storage $\texttt{store}$ and outputs an updated storage value $\texttt{store}$ as well as a group element $s \in \mathbb{Z}_N^*$.

6.1 Storage Algorithms

We assume that there exists an integer '$\texttt{levels}$' such that $T = 2^{\texttt{levels}+1} - 2$. (One could always pad T out to match this.) The key storage will be structured as a sequence of sets $S_1, \dots, S_{\texttt{levels}}$ where elements of set S_i are of the form

$$w \in \mathbb{Z}_N^*, \texttt{open} \in [1, T], \texttt{closing} \in [1, T], \texttt{count} \in [1, T].$$

Let R be the set of integers $[\texttt{open}, \texttt{open} + 2^{i-1} - 1] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1} - 1]$. Then $w = v^{\prod_{j \in T \setminus R} e_j}$. Intuitively, w is v raised to all of the e exponents except the sequence of 2^{i-1} values

[7] We expect this to be the normal mode of operation in a synchronized scheme, however, the previous schemes have the ability to sign for periods in an arbitrary order.

starting at $\texttt{open}$ and a second sequence of length $2^{i-1} - \texttt{count}$ starting at $\texttt{closing} + \texttt{count}$. When the $\texttt{StorageUpdate}$ algorithm runs for each i, it will find an element of the set S_i and help "move it forward" by incrementing its counter $\texttt{count}$ and updating w accordingly. When $\texttt{count}$ reaches 2^i the update storage algorithm removes the tuple from the set S_i at level i and then splits it into two parts and puts these in set S_{i-1}. We now describe the algorithms.

StorageInit$(\texttt{pp}, v)$ Initially, sets $S_1, \ldots, S_{\texttt{levels}}$ are empty. Then for $i = 1$ to $\texttt{levels}$ perform the following:

- Let $R = [2^i - 1, 2^{i+1} - 2]$.
- Compute $w = v^{\prod_{j \in T \setminus R} e_j}$.
- Put in S_i $(w, 2^i - 1, (2^i - 1) + 2^{i-1}, 0)$.
- Put in S_i $(w, (2^i - 1) + 2^{i-1}, 2^i - 1, 0)$.

The storage value $\texttt{store} = \big((S_1, \ldots, S_{\texttt{levels}}), \texttt{index} = 0\big)$ is output.

StorageUpdate$(\texttt{pp}, \texttt{store})$ For $i = 1$ to $\texttt{levels}$ perform the following:

- Find a tuple (if any exist) in S_i of $(w, \texttt{open}, \texttt{closing}, \texttt{count})$ with the smallest $\texttt{open}$ value.[8]
- Replace it with a new tuple $(w' = w^{e_{\texttt{closing}+\texttt{count}}}, \texttt{open}' = \texttt{open}, \texttt{closing}' = \texttt{closing}, \texttt{count}' = \texttt{count} + 1)$ where $(w', \texttt{open}', \texttt{closing}', \texttt{count}')$ is the newly added tuple .

Then for $i = \texttt{levels}$ down to 2

- Find a tuple (if any) of the form $(w, \texttt{open}, \texttt{closing}, \texttt{count} = 2^{i-1})$ in S_i.
- Remove this tuple from the set S_i.
- To the set S_{i-1} add the tuple $(w' = w, \texttt{open}' = \texttt{open}, \texttt{closing}' = \texttt{open} + 2^{i-2}, \texttt{count}' = 0)$ where $(w', \texttt{open}', \texttt{closing}', \texttt{count}')$ is the newly added tuple.
- Also add to the set S_{i-1} the tuple $(w' = w, \texttt{open}' = \texttt{open}{+}2^{i-2}, \texttt{closing}' = \texttt{open}, \texttt{count}' = 0)$.

Finally, from S_1 find the tuple $(w, \texttt{open} = \texttt{index} + 1, \texttt{closing}, 1)$. Remove this from S_1 and output $s = w$ which gives $s = v^{\prod_{j \in T \setminus \{(\texttt{index}+1)\}} e_j}$ as needed. Finally, the storage value $\texttt{store} = ((S_1, \ldots, S_{\texttt{levels}}), \texttt{index} = \texttt{index} + 1)$ is output.

6.2 Analysis

We need to show that the storage primitives give the desired correctness and performance properties. To analyze correctness and storage size we consider what the key storage state will look like for each value of $\texttt{index}$ between 0 and T. Recall that in a stored key, $\texttt{index}$ represents the number of signatures generated so far. We describe what each S_i set contains for a particular $\texttt{index}$ value — breaking things into three cases. We will refer to this as our "state description" given below.

[8] In a particular S_i there might be zero, one or two tuples. If there are two, the one with the larger $\texttt{open}$ value is ignored. Ties will not occur, as we will see from the case analysis in Section 6.2.

Case 1: $T - \texttt{index} \leq 2^i - 2$. In this case the set S_i will be empty.

Case 2: Not Case 1 and $\texttt{index} = k \cdot 2^i + r$ for $0 \leq r < 2^{i-1}$. S_i will contain two elements. The first is a tuple

$$(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i - 1, \texttt{closing} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r).$$

Where we let $R = [\texttt{open}, \texttt{open} + 2^{i-1} - 1] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1} - 1]$.

The second is a tuple

$$(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{closing} = (k+1) \cdot 2^i - 1, \texttt{count} = 0).$$

Where $R = [\texttt{open}, \texttt{open} + 2^{i-1} - 1] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1} - 1]$. (Here $\texttt{count} = 0$.)

Case 3: Not Case 1 and $\texttt{index} = k \cdot 2^i + r$ for $2^{i-1} \leq r < 2^i$. S_i has a single element. A tuple

$$(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{closing} = (k+1) \cdot 2^i - 1, \texttt{count} = r - 2^{i-1}).$$

Where $R = [\texttt{open}, \texttt{open} + 2^{i-1}] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1}]$.

Proof of State Description Accuracy

Theorem 6.1 *The above state description for variable* `index` *accurately describes the key storage state after an initial call to* $\texttt{StorageInit}(\texttt{pp}, v)$ *and* `index` *subsequent calls to* $\texttt{StorageUpdate}(\texttt{pp}, \texttt{store})$.

Proof. We begin our proof by establishing two claims about when the "pass down" operation can and cannot happen which will be used later on in the proof.

Claim 6.2 *Suppose that our state description is accurate for period* `index`. *Consider an update operation where the period moves from* `index` *to* $\texttt{index}+1$. *This will result in an tuple being "passed down" from* S_i *to* S_{i-1} *only if* $\texttt{index} + 1$ *is a multiple of* 2^{i-1}, *if anything is passed down at all.*

Proof. If $(\texttt{index}, i)$ were in Case 1, then S_i is empty and there is nothing that could be passed down. If in Case 2, then one tuple has a $\texttt{count} = r$ which is the remainder of `index` mod 2^i. It will trigger a pass down operation only when `count` increments to $\texttt{count} = 2^{i-1}$. Similarly, in Case 3 there is a tuple with $\texttt{count} = r - 2^{i-1}$. A push down operation is only triggered when it increments to 2^i which means $\texttt{index} + 1$ is a multiple of 2^{i-1}. □

Claim 6.3 *Suppose that our state description is accurate for period* `index` *and all smaller values. Further suppose that* $\texttt{index} + 1 = 0 \mod 2^i$ *for some* i *and that set* S_{i+1} *is in Case 1 at* `index`. *(I.e.* $T - \texttt{index} \leq 2^{i+1} - 2$.*) Then it will be that at period* $\texttt{index} + 1$ *we have* $T - \texttt{index} \leq 2^i - 2$ *and set* S_i *is designated as Case 1 by our description.*

Proof. Let z be the value where $T - z = 2^{i+1} - 2$ since $T = 2^{\texttt{levels}+1} - 2$ it follows that $z = y \cdot 2^{i+1}$ for some y. Also note that z must be the smallest value of `index` where $T - \texttt{index} \leq 2^{i+1} - 2$. It then follows that $z + 2^i - 1$ is the smallest value of `index` where $T - \texttt{index} \leq 2^{i+1} - 2$ AND `index` mod 2^i. Now let's consider the next value of of $\texttt{index} + 1$ which is equal to $z + 2^i$ and use it to prove that at $\texttt{index} + 1$ the set S_i is assigned to be in Case 1. Then

$$T - (\texttt{index} + 1) = T - (z + 2^i) = (T - z) - 2^i = 2^{i+1} - 2 - 2^i = 2^i - 2.$$

Then we have that at $\texttt{index}+1$ the set S_i is categorized at Case 1 (and empty) by our description. □

We now show that for each `index` if the state description was valid at `index` then it is valid at $\texttt{index}+1$. We break this into three separate claims showing that if a set S_i is in Case 1,2 and 3 respectively at `index` that in $\texttt{index}+1$ it will match the state description.

Claim 6.4 *Suppose at period* `index` *the state description is accurate and for a set* S_i *we are in Case 1 where* $T - \texttt{index} \leq 2^i - 2$ *and the set* S_i *is empty. Then at period* $\texttt{index}+1$ *the state description is accurate for set* S_i*.*

Proof. For period $\texttt{index}+1$ we have that $T - (\texttt{index}+1)$ is also $\leq 2^i - 2$ and therefore it should also be Case 1 and S_i should remain empty. The only way for it not to remain empty would be if the `StorageUpdate` algorithm "passed down" a new tuple from S_{i+1}. However, if S_i was in Case 1 for period `index` then S_{i+1} must also be and also be empty. Since S_{i+1} is empty there is nothing to pass down. □

Claim 6.5 *Suppose at period* `index` *the state description is accurate and for a set* S_i *we are in Case 2 where* $\texttt{index} = k \cdot 2^i + r$ *for* $0 \leq r < 2^{i-1}$*. Then at period* $\texttt{index}+1$ *the state description is accurate for set* S_i*.*

Proof. First consider the subcase where $r \neq 2^{i-1} - 1$ which should keep S_i in Case 2 on period $\texttt{index}+1$. We will verify this. Since at period `index` we are in Case 2 there are two tuples in S_i where the one with the smaller `open` value is of the form $(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i - 1, \texttt{closing} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r)$. The update algorithm will increment `count` to $r+1$ and update w to $w = w^{e_{\texttt{closing}+\texttt{count}}}$ which gives the needed form to remain in Case 2. The second tuple will is of the form $(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{closing} = (k+1) \cdot 2^i - 1, \texttt{count} = 0)$. The update algorithm will not modify it as the other tuple had the smaller `open` value. Thus it remains the same which matches the behavior for S_i remaining in Case 2. Finally, we need to check that no new tuples are passed down from S_{i+1}. This follows from the fact (Claim 6.2) that `index` mod $2^i = r \neq 2^i - 1$ and that a pushdown would only happen as `index` transfers to being a multiple of 2^i.

We now consider the subcase where $r = 2^{i-1} - 1$ at `index` and thus at $\texttt{index}+1$ we should be moving into Case 3. In this subcase the set S_i begins with two tuples with one of the form $(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i - 1, \texttt{closing} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r = 2^{i-1} - 1)$. The update operation will first modify the tuple to a new `count` value of $\texttt{count} = 2^{i-1}$. This will trigger the pushdown operation to move the tuple out of S_i. It then leaves it with one tuple of the needed form which transitions S_i to Case 3 as needed. Again no new elements are pushed onto S_i from S_{i+1} due to Claim 6.2. □

Claim 6.6 *Suppose at period* `index` *the state description is accurate and for a set* S_i *we are in Case 3 where* $\texttt{index} = k \cdot 2^i + r$ *for* $2^{i-1} \leq r < 2^i$ *for some* k*. Then at period* $\texttt{index}+1$ *the state description is accurate for set* S_i*.*

Proof. We first focus on the subcase where $r \neq 2^i - 1$ and thus at $\texttt{index}+1$ we want to verify that we stay in Case 3. Initially there is one tuple of the form $(w = v^{\prod_{j \in T \setminus R} e_j}, \texttt{open} = (k+1) \cdot 2^i -$

$1+2^{i-1}, \texttt{closing} = (k+1)\cdot 2^i - 1, \texttt{count} = r - 2^{i-1})$. The update algorithm will increment $\texttt{count}$ to $r+1$ and update w to $w = w^{e_{\texttt{closing}+\texttt{count}}}$ which gives the needed form to remain in Case 3. As before no new tuples will be added since $\texttt{index}+1 \mod 2^i \neq 0$.

We end by considering the subcase where $r = 2^i - 1$. In this subcase there is initially a single tuple with a $\texttt{count}$ value of $\texttt{count} = 2^{i-1} - 1$. The update algorithm will increment this $\texttt{count}$ which triggers its removal from the set. What remains to be seen is whether a new element is added or if it becomes empty.

We now consider two possibilities. If $T - (\texttt{index}+1) \leq 2^i - 2$, then our description states that set S_i should enter Case 1 on $\texttt{index}+1$. It is easy to see that if this is true that the set S_{i+1} was already Case 1 and empty on $\texttt{index}$ and nothing new will be added so the set S_i is empty as needed.

The somewhat trickier case is when $T-(\texttt{index}+1) > 2^i - 2$. Here we need to verify that the set S_i ends up in Case 2 with the appropriate tuple at $\texttt{index}+1$. First, since $\texttt{index}+1 \mod 2^i = 0$ we can apply Claim 6.3. It states that if set S_{i+1} were in Case 1 (empty) at $\texttt{index}$ then set S_i would be in Case 1 for $\texttt{index}+1$. Since this is not the case, we have that S_{i+1} must be non empty and in Case 2 or 3.

If S_{i+1} started in Case 2 at $\texttt{index}$, it initially has a tuple of the form:

$$(w = v^{\prod_{j\in T\setminus R} e_j}, \texttt{open} = (\tilde{k}+1)\cdot 2^{i+1} - 1, \texttt{closing} = (\tilde{k}+1)\cdot 2^{i+1} - 1 + 2^i, \texttt{count} = 2^i - 1).$$

Where we let $R = [\texttt{open}, \texttt{open}+2^i-1] \cup [\texttt{closing}+\texttt{count}, \texttt{closing}+2^i-1]$. Note by the description $\texttt{index} = 2^{i+1}\tilde{k} + 2^i - 1$. After the update algorithm has its first pass, $\texttt{count}$ is incremented to 2^i and an exponentiation is done that updates w where it is now for $R = [\texttt{open}, \texttt{open}+2^i-1]$ as the second half of the range falls off with the new count value. The update algorithm then removes this tuple from S_{i+1} and creates two new tuples from it. One with an $\texttt{open}' = \texttt{open}$ and $\texttt{closing}' = \texttt{open}+2^i$; the second with $\texttt{open}' = \texttt{open}+2^i$ and $\texttt{closing}' = \texttt{open}$.

To verify correctness recall that $\texttt{index} = 2^i k + 2^i - 1$ and $\texttt{index} = 2^{i+1}\tilde{k} + 2^i - 1$. It follows that $k = 2\cdot\tilde{k}$. Second, $\texttt{index}+\texttt{1} = 2^i \cdot k'$ where $k' = k+1$. To match the description for $\texttt{index}+\texttt{1}$ we must have that the first tuple created has an $\texttt{open}'$ value of $\texttt{open}' = (k'+1)2^i - 1$. Plugging in terms:

$$(k'+1)2^i - 1 = (k+1+1)2^i - 1 = (2\tilde{k}+2)2^i - 1 = (\tilde{k}+1)2^{i+1} - 1.$$

However, this is exactly the value it inherited from $\texttt{open}$ as needed.

The argument that the right tuple is inherited when set S_{i+1} is in Case 3 proceeds in almost the same way as above.

□

The proof of our theorem now comes via induction. The accuracy of the state description for $\texttt{index} = 0$ can be verified by inspection. We can prove the rest by induction on $\texttt{index}$. For any $\texttt{index}$ the accuracy of the description $\texttt{index}+1$ follows from its accuracy on period $\texttt{index}$. In particular, our previous three claims show that for any i if the state S_i is accurate in period $\texttt{index}$ then after the update algorithm executes, S_i will be accurate in period $\texttt{index}+1$ as well. □

Computational and Storage Efficiency Analyzing the running time for these storage operations is straightforward. We have that $\texttt{levels} = \lfloor \lg T \rfloor$. In each storage update operation there is at each level at most one prime search operation and at most one exponentiation. This comes from the fact that for each i the algorithm updates a single set element — the one with the smallest `open` value (if any). Therefore the number of prime searches and exponentiations is bounded by $\lg(T)$ as desired.

The above state description immediately gives us the storage efficiency we desire. There are at most $\lg(T)$ sets i which have at most two tuples. Each tuple has a single group element. As written, a tuple also has three (small) integers (of value at most T), although a program could drop these because they can be inferred from `index`, so we will not count them in our Section 8 analysis.

Sample Snapshot of Storage To aid the reader, we provide an example of the storage states for $\texttt{levels} = 3$ and $T = 2^{\texttt{levels}+1} - 2 = 2^4 - 2 = 14$ in Figure 1. This example shows the states after updates; it does not show any intermediate states during an update operation. The example gives just the `open`, `closing` and `count` values. The prior section describes how the corresponding group element w is computed based on these values (see the description of R as the range of indices of e_i values excluded from the product in the exponent.) Initially, we have sets $S_1, \ldots, S_{\texttt{levels}=3}$ that are empty. The values at $\texttt{index} = 0$ show the states after running `StorageInit`. The values at $\texttt{index} > 0$ show the state after a call to `StorageUpdate`.

6.3 Using the Storage Primitives and Optimizations

We can use the storage primitive above to modify our signing algorithm and key storage of Section 4. We describe two slightly different methods to do this.

Method 1 The Setup algorithm will run as before and output the core public parameters as $\mathsf{pp} = (T, N, g, Y, K)$. However, it will also run $\texttt{StorageInit}(\mathsf{pp}, g)$ which outputs a value `store` which is appended to the public parameters.

The secret key algorithm will choose random integers $u_0, u_1, \ldots, u_k$ in $[1, N]$. It sets the secret key as $sk = (u_0, u_1, \ldots, u_k)$ and the public key $pk = (U_0, U_1, \ldots, U_k)$ where $U_j = Y^{u_j} = g^{u_j \prod_{i \in T} e_i}$, for $j = 0$ to k. Note all of this is identical to the Section 4 scheme. However, it additionally appends `store` from the public parameters to its secret key. The `store` is the part of the secret key that will be modified at each signing.

During each the t-th signing step, it will call $\texttt{StorageUpdate}(\mathsf{pp}, \texttt{store}_{t-1})$ and as output get a new storage value $\texttt{store}_t$ that is uses to replace the previous one as well as $J = Y^{1/e_t}$. It uses this to sign by computing:

$$\sigma = J^{u_0} \prod_{j=1}^{k} J^{u_j \cdot m_j} = \big(U_0 \prod_{j=1}^{k} U_j^{m_j}\big)^{1/e_t} \pmod{N}.$$

Method 2 This will be similar to Method 1 except that instead of raising to the $u_0, \ldots, u_k$ values at the end of signing the algorithm will keep $k+1$ parallel copies of storage that already have each respective u_i exponent raised. The description below will need to slightly "break into" the abstraction that we gave earlier.

Setup will run as before and output the core public parameters as $\mathsf{pp} = (T, N, g, Y, K)$. However, it will also run $\texttt{StorageInit}(\mathsf{pp}, g)$ which outputs a value `store` which is appended to the public parameters.

The secret key algorithm will choose random integers $u_0, u_1, \ldots, u_k$ in $[1, N]$. It sets the public key $pk = (U_0, U_1, \ldots, U_k)$ where $U_j = Y^{u_j} = g^{u_j \prod_{i \in T} e_i}$, for $j = 0$ to k (as in the Section 4 scheme).

`index`	Set S_1			Set S_2			Set S_3		
	`open`	`closing`	`count`	`open`	`closing`	`count`	`open`	`closing`	`count`
0	1 2	2 1	0 0	3 5	5 3	0 0	7 11	11 7	0 0
1	2	1	0	3 5	5 3	1 0	7 11	11 7	1 0
2	3 4	4 3	0 0	5	3	0	7 11	11 7	2 0
3	4	3	0	5	3	1	7 11	11 7	3 0
4	5 6	6 5	0 0	7 9	9 7	0 0	11	7	0
5	6	5	0	7 9	9 7	1 0	11	7	1
6	7 8	8 7	0 0	9	7	0	11	7	2
7	8	7	0	9	7	1	11	7	3
8	9 10	10 9	0 0	11 13	13 11	0 0			
9	10	9	0	11 13	13 11	1 0			
10	11 13	12 11	0 0	12	11	0			
11	12	11	0	13	11	1			
12	13 14	14 13	0 0						
13	14	13	0						
14									

Figure 1: Storage State Example for `levels` $= 3$, $T = 14$. See snapshot description.

For $j = 0$ to k it computes $\texttt{store}^{(j)}$ by taking each of the group elements in `store` and raising it to u_j. This process effectively changes `store` from being a storage of $v = g$ to being a storage of $v_j = g^{u_j}$ for the respective u_j. Note that each conversion takes $2 \cdot \texttt{levels}$ exponentiations since there are $2 \cdot \texttt{levels}$ group elements per storage.

During each the t-th signing step, for each $j \in [0, k]$ it will call $\texttt{StorageUpdate}(\texttt{pp}, \texttt{store}_{t-1}^{(j)})$ and as output get a new storage value $\texttt{store}_t^{(j)}$ that is uses to replace the previous one as well as $J_j = U_j^{1/e_t}$. It uses these to sign by computing:

$$\sigma = J_0 \prod_{j=1}^{k} J_j^{m_j} = \big(U_0 \prod_{j=1}^{k} U_j^{m_j}\big)^{1/e_t} \pmod{N}.$$

One final efficiency note: in the scheme above on the update operation will perform `levels` prime searches for each of the $k + 1$ stores. (By prime search we mean computing the relevant e_i

values needed in update.) This gives $(k+1) \cdot \texttt{levels}$ total prime searches. However, each of these stores will be computing the same e values. Thus if we slightly break into the abstraction then one can do only `levels` total prime searches by sharing that part of the computation across all $k+1$ storage updates.

7 Identity-Based Aggregation from RSA

In Appendix A, we provide the definition for synchronized identity-based aggregate signatures. We now give a construction based on the RSA assumption.

Setup$(1^\lambda, 1^T)$ The setup algorithm chooses an integer $N = pq$ as the product of two safe primes where $p-1 = 2p'$ and $q-1 = 2q'$, such that $2^\lambda < \phi(N) < 2^{\lambda+1}$. The scheme assumes a hash function (modeled as a random oracle) $G : \mathcal{I} \to \mathbb{Z}_N^{*(k+1)}$. It also uses the hash function $H : [1, T] \to \{0,1\}^{\lambda+1}$ with key K as specified in Section 4. It computes:

$$D = \prod_{i=1}^{T} H_K(i)^{-1} \mod \phi(N).$$

It publishes the public parameters as $\mathsf{pp} = (T, N, K)$ and we assume all parties have access to G. The master secret key includes the factorization of N and the value D.

Extract(msk, ID) The algorithm computes $(U_0, \ldots, U_k) \leftarrow G(ID)$. For $i = 1$ to k, it computes $d_i = U_i^D \mod N$. It returns the secret key as $sk = (d_0, d_1, \ldots, d_k)$.

Sign$(\mathsf{pp}, sk_{ID}, M, t)$ The signing algorithm takes as input a time period $1 \leq t \leq T$ and an $L = (\ell k)$-bit message $M = m_1|m_2|\ldots|m_k$, where each m_i contains ℓ-bits and these are concatenated together to form M. It computes the primes $(e_1, \ldots, e_T)$ from pp and then outputs

$$\sigma = \big(d_0 \prod_{j=1}^{k} d_j^{m_j}\big)^{\prod_{i \in T \setminus \{t\}} e_i} = \big(U_0 \prod_{j=1}^{k} U_j^{m_j}\big)^{1/e_t} \pmod N.$$

Verify$(\mathsf{pp}, ID, M, t, \sigma)$ Let $M = m_1|m_2|\ldots|m_k$ and $G(ID) = (U_0, \ldots, U_k)$ The algorithm computes the prime e_t from pp. Output 1 if $1 \leq t \leq T$ and $\sigma^{e_t} \stackrel{?}{=} U_0 \prod_{i=1}^{k} U_i^{m_i}$ or 0 otherwise.

Aggregate$(\mathsf{pp}, t, (ID_1, M_1, \sigma_1), \ldots, (ID_n, M_n, \sigma_n))$ As before, $\sigma_{agg} = \prod_{j=1}^{n} \sigma_j \pmod N$.

AggVerify$(\mathsf{pp}, t, (ID_1, \ldots, ID_n), (M_1, \ldots, M_n), \sigma_{agg})$ As before, output 1 if and only if all inputs are in the correct range, each identity is unique and $\sigma_{agg}^{e_t} \stackrel{?}{=} \prod_{j=1}^{n}(U_{j,0} \prod_{i=1}^{k} U_{j,i}^{m_{j,i}})$ where here $G(ID_i) = (U_{i,0}, \ldots, U_{i,k})$.

Scheme	Signing Operations				
	$\mathbb{P}$	$\mathbb{E}_N$	$\mathbb{E}_{\lvert e\rvert}$	$\mathbb{E}_\ell$	$\mathbb{M}$
Section 4	$T-1$	$k+1$	$T-1$	k	k
Section 5 ($a=\sqrt{T}$)	$\sqrt{T}-1$	$k+1$	$\sqrt{T}-1$	k	k
Section 5 ($a=1$)	0	$k+1$	0	k	k
Section 6 Method 1	$\lg(T)$	$k+1$	$\lg(T)$	k	k
Section 6 Method 2	$\lg(T)$	0	$(k+1)\lg(T)$	k	k

Figure 2: Signing Operations Evaluation. Let the modulus be N. Let $\mathbb{P}$ be the time for function H_K to output a prime of $|e|$ bits, $\mathbb{E}_j$ be the time to perform a j-bit modular exponentiation, and $\mathbb{M}$ be the time to perform a modular multiplication. For the Section 6, we round up and treat $\lg T \approx$ `levels`. For that scheme via Method 2, the results of the prime search from the first store are shared with all other stores.

Operation	$\mathbb{P}_{257}$	$\mathbb{P}_{80}$	$\mathbb{E}_{2048}$	$\mathbb{E}_{257}$	$\mathbb{E}_{256}$	$\mathbb{E}_{80}$	$\mathbb{E}_{32}$	$\mathbb{M}$
Time (ms)	0.975	0.311	4.604	0.670	0.629	0.235	0.094	0.00091

Figure 3: Time recorded in milliseconds for the above operations are averaged over 1,000 iterations for a 2048-bit modulus using NTL v10.5.0 on a modern laptop. Let $\mathbb{P}_x$ denote an x-bit prime search, $\mathbb{E}_x$ be an x-bit modular exponentiation, and $\mathbb{M}$ be a modular multiplication.

Remarks. We remark that the same performance enhancements explored in Sections 5 and 6 apply here. For simplicity, we present the identity-based version only for the scheme in Section 4.

Theorem 7.1 *If the RSA assumption (as stated in Assumption 3.1) holds, F is a secure pseudorandom function and G is modeled as a random oracle, then the above identity-based synchronized aggregate signature construction is existentially unforgeable under an adaptive chosen message attack.*

Proof of this theorem appears in Appendix A.1.

8 Performance Evaluation

We now analyze the performance of the various RSA-based aggregate signature schemes presented in this work. In particular we consider: our core signature scheme of Section 4, our scheme with $\approx \sqrt{(T)}$ storage and signing time of Section 5, our "big storage for fast signing" scheme also of Section 5 and our scheme of $\approx \lg(T)$ storage and signing of Section 6 via two different methods of implementing signing (which may out perform each other based on the selection of various implementation parameters). The scheme of Section 4 has similar performance to that of Section 5 when $a = T$ and therefore we do not separately analyze it.

For each scheme, we first evaluate its run-time performance with a signing algorithm operations count in Figure 2. We then proceed to inspect its practical performance using a 2048-bit RSA modulus and a 256-bit message (the latter corresponding to an output of SHA-256). In Figure 4, we evaluate each scheme with each of the following parameters: 1 message chunk size of 256 bits,

Scheme	Parameters			Time when $T =$					
	k	ℓ	$\lvert e\rvert$	2^{12}	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}
Section 4	1	256	257	6.7s	1.8m	28.7m	7.7h	5.1d	81.7d
	8	32	80	2.3s	35.8s	9.5m	2.5h	1.7d	27.1d
	256	1	80	3.4s	37.0s	9.6m	2.5h	1.7d	27.1d
Section 5 $(a = \sqrt{T})$	1	256	257	113.4ms	0.4s	1.7s	6.7s	27.0s	1.8m
	8	32	80	76.6ms	0.2s	0.6s	2.3s	9.0s	35.8s
	256	1	80	1.2s	1.3s	1.7s	3.4s	10.1s	36.8s
Section 5 $(a = 1)$	1	256	257	9.8ms	9.8ms	9.8ms	9.8ms	9.8ms	9.8ms
	8	32	80	42.2ms	42.2ms	42.2ms	42.2ms	42.2ms	42.2ms
	256	1	80	1.2s	1.2s	1.2s	1.2s	1.2s	1.2s
Section 6 Method 1	1	256	257	29.6ms	36.1ms	42.7ms	49.3ms	55.9ms	62.5ms
	8	32	80	48.8ms	50.9ms	53.1ms	55.3ms	57.5ms	59.7ms
	256	1	80	1.2s	1.2s	1.2s	1.2s	1.2s	1.2s
Section 6 Method 2	1	256	257	28.4ms	37.7ms	47.0ms	56.2ms	65.4ms	74.7ms
	8	32	80	29.9ms	39.6ms	49.3ms	59.1ms	68.8ms	78.5ms
	256	1	80	0.7s	1.0s	1.2s	1.5s	1.7s	1.9s

Figure 4: Signing Time Evaluations for 90 different performance points; here N is 2048 bits. Times are calculated by taking the average time for an operation (see Figure 3) and summing up the total times of each operation (see Figure 2). Let ms denote milliseconds, s denote seconds, m denote minutes, h denote hours, and d denote days.

8 message chunks of 32 bits and 256 messages chunks of 1 bit. When message chunks are 256 bits, we use 257-bit prime e values and for chunks of size 32 bits or 1 bit we consider 80-bit e values. Here we make sure that the size of the RSA primes are at least as big as the message chunks, but let them fall no further than 80 bits to avoid collisions.[9] These evaluations will be considered for a maximum number of periods of $T \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}, 2^{32}\}$. Technically, for the log scheme the numbers of time periods is $T = 2^{\texttt{levels}+1} - 2$, however for the sake of these comparisons we will ignore the small constants.

To perform the timing evaluations in Figure 4, we utilized the high-performance NTL number theory library in C++ v10.5.0 by Victor Shoup [37]. Averaged over 1000 iterations, we measured the cost of a prime search of the relevant size as well as the time to compute modular multiplications and modular exponentiations for the relevant exponent sizes using a 2048-bit RSA modulus. We took all time measurements on an early 2015 MacBook Air with a 1.6 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory. These timing results are recorded in Figure 3.

We next report on the signer's storage space requirements in Figure 5 for all of these combinations. And in Figure 6, we show how to view T in practical terms for how often one can issue signatures according to the synchronized restrictions over the lifespan of a private key.

[9] We remark that the parameters given for this evaluation do not have a total correspondence to the scheme description. For example, using 80-bit e values will technically require a variant of the RSA assumption with smaller exponents. And we do not attempt to set the modulus size to match the security loss of our reduction. (It is unknown whether this loss can actually be utilized by an attacker or not.) Our focus here is to give the reader a sense of the relative performance of the scheme variants for parameters that might be used in practice.

Scheme	SK Elements	Param.	Size when $T =$					
	$\mathbb{Z}_N$	k	2^{12}	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}
S4	$k+1$	1	0.5K	0.5K	0.5K	0.5K	0.5K	0.5K
		8	2.3K	2.3K	2.3K	2.3K	2.3K	2.3K
		256	64.3K	64.3K	64.3K	64.3K	64.3K	64.3K
S5 ($a = \sqrt{T}$)	$(k+1) + \sqrt{T}$	1	16.5K	64.5K	256.5K	1.0M	4.0M	16.0M
		8	18.3K	66.3K	258.3K	1.0M	4.0M	16.0M
		256	80.3K	128.3K	320.3K	1.1M	4.1M	16.1M
S5 ($a = 1$)	$(k+1) + T$	1	1.0M	16.0M	256.0M	4.0G	64.0G	1.0Tb
		8	1.0M	16.0M	256.0M	4.0G	64.0G	1.0Tb
		256	1.1M	16.1M	256.1M	4.0G	64.0G	1.0Tb
S6 Method 1	$(k+1) + 2\lg T$	1	6.5K	8.5K	10.5K	12.5K	14.5K	16.5K
		8	8.3K	10.3K	12.3K	14.3K	16.3K	18.3K
		256	70.3K	72.3K	74.3K	76.3K	78.3K	80.3K
S6 Method 2	$2(k+1)\lg T$	1	12.0K	16.0K	20.0K	24.0K	28.0K	32.0K
		8	54.0K	72.0K	90.0K	108K	106K	144K
		256	1.5M	2.0M	2.5M	3.0M	3.5M	4.0M

Figure 5: Private Key Size Evaluation. Here the modulus N is 2048 bits. The above numbers are rounded to show one decimal point. Let K denote a kilobyte (2^{10} bytes), M a megabyte (2^{20} bytes), G a gigabyte (2^{30} bytes), and Tb a terabyte (2^{40} bytes). Any of the schemes that compute primes during Signing (all but Section 5 when $a = 1$), could instead choose to speed up signing by additionally storing those values at an additional storage cost of T elements of $\mathbb{Z}_{|e|}$. All but the last scheme include $k+1$ elements that are the randomization factors $u_0, \ldots, u_k \in [1, N]$; this space could be shrunk by re-computing these from a PRF.

Some Conclusions As expected the initial core scheme of Section 4 is much too costly for signing. Even for $T = 2^{20}$ (where one signature is permitted every 5 minutes for 10 years), it takes roughly 10 minutes to sign a single message, so the processor we took these measurements on could not "break even" by keeping up with the modest pace of one signature every 5 minutes using the base scheme. At larger time periods, the signing time moves into days. One noticeable aspect is that the $k = 1$ (where k is the number of message chunks) time measurements are about a factor of three greater than when $k \in \{8, 256\}$ for this scheme and the square root one. This is due to the cost difference of searching for and raising to 257-bit primes versus 80-bit primes which dominate these schemes.

The square root tradeoff certainly does better, but cannot break even (on the processor measured) once we hit $T = 2^{28}$. Additionally, the keys are somewhat large on the order of a few megabytes. This could be an issue if we would want to store several keys or a single key on a low memory device.

On the other end of the spectrum when setting $a = 1$, we get relatively fast signatures. Here things flip where it is significantly more expensive to sign for $k = 256$ than $k \in \{1, 8\}$. The reason is that at this point the cost of raising to the u_i values now dominates the computation — whereas in the earlier schemes it was dominated by raising to the e_i values. The main downside of this setting is that the key sizes are huge — breaking into the terabyte range for $T = 2^{32}$.

Setting of T	Frequency of Signatures
2^{12}	76,992 sec (≈ one per day)
2^{16}	4,812 sec (≈ one every 1.5 hours)
2^{20}	300 sec (≈ one every 5 minutes)
2^{24}	19 sec
2^{28}	1.2 sec
2^{32}	0.07 sec (≈ ten per second)

Figure 6: Approximate view of how to select T based on how often an application needs the ability to issue signatures during a key's 10-year lifespan. (One can approximate a 20-year key lifespan by cutting the above frequencies in half.)

We finally move to our log scheme of Section 6 where we start with Method 1. It scales well with the number of time periods where even for $T = 2^{32}$ it is only about 60ms for $k \in \{1, 8\}$. For $k = 256$ the time is again dominated by the raising to the u_i values at the end. Also, the private keys can be kept in the range of ten to twenty kilobytes for lower k values. (We note that for $k = 256$ one possibility is that the u_i values could be generated from a pseudo random function which could lower the key storage cost.) The second method of using the log storage is more costly in terms of key storage cost. Its performance in signing time is slightly better for smaller values of T, but for values higher than 2^{20} turns worse. For this reason the first method seems to perform better overall than the second.

Altogether, the log storage solution (of Section 6 using Method 1) offers practical time/space costs and appears to provide the best overall practical performance of all schemes analyzed.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and Joseph Ayo Akinyele for implementation discussions.

References

[1] Jae Hyun Ahn, Matthew Green, and Susan Hohenberger. Synchronized aggregate signatures: new definitions, constructions and applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS*, pages 473–484, 2010.

[2] Anonymous. Increasing anonymity in bitcoin, 2013. See discussion and link to paper at `https://bitcointalk.org/index.php?topic=1377298.0`.

[3] Ali Bagherzandi and Stanislaw Jarecki. Identity-Based Multi-Signatures based on RSA. In *PKC '10*, volume 6056 of LNCS, pages 480–498, 2010.

[4] Rachid El Bansarkhani, Mohamed Saied Emam Mohamed, and Albrecht Petzoldt. MQSAS - A multivariate sequential aggregate signature scheme. In *Information Security - 19th International Conference, ISC*, pages 426–439, 2016.

[5] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *45th Symposium on Foundations of Computer Science (FOCS)*, pages 186–195. IEEE Computer Society, 2004.

[6] Niko Baric and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology - EUROCRYPT*, pages 480–494, 1997.

[7] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In *ICALP '07*, volume 4596 of LNCS, pages 411–422, 2007.

[8] Mihir Bellare and Gregory Neven. Identity-Based Multi-signatures from RSA. In *CT-RSA '07*, volume 4377 of LNCS, pages 145–162, 2007.

[9] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *Advances in Cryptology - EUROCRYPT*, pages 274–285, 1993.

[10] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Public Key Cryptography - PKC*, pages 31–46, 2003.

[11] Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In *ACM Conference on Computer and Communications Security (CCS)*, pages 276–285, 2007. Updated version available at `http://www.cc.gatech.edu/~amoneill/bgoy.html`.

[12] Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. *J. ACM*, 48(4):702–722, 2001.

[13] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT '03*, volume 2656 of LNCS, pages 416–432, 2003.

[14] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. A survey of two signature aggregation techniques. *RSA Cryptobytes*, 6(2):1–9, 2003.

[15] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations. *Inf. Comput.*, 239:356–376, 2014.

[16] Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security*, 3(3):161–185, 2000.

[17] Marc Fischlin, Anja Lehmann, and Dominique Schröder. History-free sequential aggregate signatures. In *Security and Cryptography for Networks - SCN*, pages 113–130, 2012.

[18] Eduarda S. V. Freire, Dennis Hofheinz, Kenneth G. Paterson, and Christoph Striecks. Programmable hash functions in the multilinear setting. In *Advances in Cryptology - CRYPTO*, pages 513–530, 2013.

[19] Craig Gentry and Zulfikar Ramzan. Identity-based aggregate signatures. In *Public Key Cryptography '06*, volume 3958 of LNCS, pages 257–273, 2006.

[20] Xinshun Guo and Zhiwei Wang. An efficient synchronized aggregate signature scheme from standard RSA assumption. *International Journal of Future Generation Communication and Networking*, 7, No. 3:229–240, 2014.

[21] Dennis Hofheinz, Tibor Jager, and Eike Kiltz. Short signatures from weaker assumptions. In *Advances in Cryptology - ASIACRYPT*, pages 647–666, 2011.

[22] Susan Hohenberger, Venkata Koppula, and Brent Waters. Universal signature aggregators. In *Advances in Cryptology - EUROCRYPT*, pages 3–34, 2015.

[23] Susan Hohenberger, Amit Sahai, and Brent Waters. Full domain hash from (leveled) multilinear maps and identity-based aggregate signatures. In *Advances in Cryptology - CRYPTO*, pages 494–512, 2013.

[24] Susan Hohenberger and Brent Waters. Short and stateless signatures from the RSA assumption. In *CRYPTO '09*, volume 5677 of LNCS, pages 654–670, 2009.

[25] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In *Advances in Cryptology - CRYPTO*, pages 332–354, 2001.

[26] Kwangsu Lee, Dong Hoon Lee, and Moti Yung. Sequential aggregate signatures made shorter. In *Applied Cryptography and Network Security (ACNS)*, pages 202–217, 2013.

[27] Kwangsu Lee, Dong Hoon Lee, and Moti Yung. Sequential aggregate signatures with short public keys: Design, analysis and implementation studies. In *Public-Key Cryptography - PKC*, pages 423–442, 2013.

[28] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In *EUROCRYPT '06*, volume 4004 of LNCS, pages 465–85, 2006. Full version at `http://cseweb.ucsd.edu/~hovav/dist/agg-sig.pdf`.

[29] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In *EUROCRYPT '04*, volume 3027 of LNCS, pages 74–90, 2004.

[30] Di Ma and Gene Tsudik. Extended abstract: Forward-secure sequential aggregate authentication. In *IEEE Symposium on Security and Privacy (S&P 2007)*, pages 86–91, 2007.

[31] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.

[32] Gregory Neven. Efficient sequential aggregate signed data. In *EUROCRYPT '08*, volume 4965 of LNCS, pages 52–69, 2008.

[33] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

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

[35] Markus Rückert and Dominique Schröder. Aggregate and verifiably encrypted signatures from multilinear maps without random oracles. In *Advances in Information Security and Assurance (ISA)*, pages 750–759, 2009.

[36] S. Sharmila Deva Selvi, S. Sree Vivek, and C. Pandu Rangan. Deterministic identity based signature scheme and its application for aggregate signatures. In *Information Security and Privacy - 17th Australasian Conference, ACISP*, pages 280–293, 2012.

[37] Victor Shoup. NTL: A Library for doing Number Theory, v10.5.0, 2017. Available at `http://www.shoup.net/ntl/`.

A Identity-Based Synchronized Aggregate Signatures

Definition A.1 (Identity-Based Synchronized Aggregate Signatures [19]) *An identity-based synchronized aggregate signature scheme for a bounded number of periods, identity space $\mathcal{I}(\cdot)$, and message space $\mathcal{M}(\cdot)$ is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{Extract}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$ *such that*

Setup$(1^\lambda, 1^T)$ *: On input the security parameter λ and the period bound T, the setup algorithm outputs public parameters* pp *and a master secret key msk.*

Extract(msk, ID) *: On input the master secret key msk and an identity $ID \in \mathcal{I}(\lambda)$, the key extraction algorithm outputs a secret key sk.*

Sign(pp, sk, M, t) *: On input the public parameters* pp*, the signing algorithm takes in a secret key sk, a message $M \in \mathcal{M}(\lambda)$, the current period $t \leq T$, and produces a signature σ.*

Verify$(\mathsf{pp}, ID, M, t, \sigma)$ *: On input the public parameters* pp*, the verification algorithm takes in an identity ID, a message M, a period t and a purported signature σ, and returns 1 if and only if the signature is valid and $t \leq T$, and 0 otherwise.*

Aggregate$(\mathsf{pp}, t, (ID_1, M_1, \sigma_1), \ldots, (ID_n, M_n, \sigma_n))$ *: On input the public parameters* pp*, a period t, a sequence of identities $(ID_1, \ldots, ID_n)$, messages $(M_1, \ldots, M_n)$, and purported signatures $(\sigma_1, \ldots, \sigma_n)$ for period $t \leq T$, it outputs an aggregate signature σ_{agg} or error message $\perp$.*

AggVerify$(\mathsf{pp}, t, (ID_1, \ldots, ID_n), (M_1, \ldots, M_n), \sigma_{agg})$ *: On input the public parameters* pp*, a period t, a sequence of public keys $(ID_1, \ldots, ID_n)$ and messages $(M_1, \ldots, M_n)$, and a purported aggregate signature σ_{agg}, the aggregate-verification algorithm outputs 1 if and only if σ_{agg} is a valid aggregate signature and $t \leq T$, and 0 otherwise.*

Efficiency *We require that the setup algorithm run in time polynomial in its inputs and all other algorithms run in time polynomial in λ, T.*

Correctness *Let* $\mathsf{poly}(x)$ *denote the set of polynomials in x. In addition to the standard correctness properties of the base identity-based signature scheme, for an indentity-based synchronized aggregation scheme, the correctness requirements on* $\mathsf{Aggregate}$ *and* $\mathsf{AggVerify}$ *stipulate that for all $\lambda \in \mathbb{N}$, $T \in \mathsf{poly}(\lambda), n \in \mathsf{poly}(\lambda)$, $\mathsf{pp} \in \mathsf{Setup}(1^\lambda, 1^T)$, $ID_1, \ldots, ID_n \in \mathcal{I}(\lambda)$, $sk_i \in \mathsf{Extract}(msk, ID_i)$,*

$1 \leq t \leq T$, $M_i \in \mathcal{M}(\lambda)$, $\sigma_i \in \mathsf{Sign}(sk_i, ID_i, M_i, t)$ and $\sigma_{agg} \in \mathsf{Aggregate}(\mathsf{pp}, t, (ID_1, M_1, \sigma_1), \ldots, (ID_n, M_n, \sigma_n))$, it holds that

$$\mathsf{AggVerify}(\mathsf{pp}, t, (ID_1, \ldots, ID_n), (M_1, \ldots, M_n), \sigma_{agg}) = 1.$$

Unforgeability The definition uses the following game between a challenger and an adversary $\mathcal{A}$ for a given scheme $\Pi = (\mathsf{Setup}, \mathsf{Extract}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$, security parameter λ, identity space $\mathcal{I}(\lambda)$ and message space $\mathcal{M}(\lambda)$:

Setup: The adversary sends 1^T to the challenger, who then runs $\mathsf{Setup}(1^\lambda, 1^T)$ to obtain the public parameters pp and the master secret msk. The adversary is sent pp.

Queries: At any time after Setup, the adversary can adaptively choose an identity ID and request the secret key for this identity. For each period t starting with 1 and incrementing up to T and any identity $ID \in \mathcal{I}$, the adversary can also request one signature for ID on any message of its choice $M_t \in \mathcal{M}$, or it can choose to skip that period. The challenger responds to a query for M_t under ID during period $t \in [1, T]$ as $\mathsf{Sign}(\mathsf{pp}, sk_{ID}, M_t, t)$.

Output: Let α be a function mapping integers to $[1, T]$. Eventually, the adversary outputs a tuple $(t, (ID_1, \ldots, ID_k), (M'_1, \ldots, M'_k), \sigma_{agg})$ and wins the game if:

1. $1 \leq t \leq T$; and
2. all $ID_i \in \mathcal{I}$ and $M'_i \in \mathcal{M}$; and
3. there exists an $z \in [1, k]$ such that (ID_z, M'_z) is non-trivial (i.e., during the Queries phase $\mathcal{A}$ did not request the private key for ID_z and did not request a signature including the pair (ID_z, M'_z)); and
4. $\mathsf{AggVerify}(\mathsf{pp}, t, (ID_1, \ldots, ID_k), (M'_1, \ldots, M'_k), \sigma_{agg}) = 1$, where $1 \leq k \leq T$.

We define $\mathsf{SigAdv}_{\mathcal{A},\Pi,\mathcal{I},\mathcal{M}}(\cdot)$ to be the probability that the adversary $\mathcal{A}$ wins in the above game with scheme Π for identity space $\mathcal{I}$, message space $\mathcal{M}$ and security parameter λ taken over the coin tosses made by $\mathcal{A}$ and the challenger.

Definition A.2 (Unforgeability) *An identity-based synchronized aggregate signature scheme Π for identity space $\mathcal{I}$, message space $\mathcal{M}$ is existentially unforgeable under an adaptive chosen message attack if for all sufficiently large $\lambda \in \mathbb{N}$ and all probabilistic polynomial-time in λ adversaries $\mathcal{A}$, there exists a negligible function* negl, *such that* $\mathsf{SigAdv}_{\mathcal{A},\Pi,\mathcal{I},\mathcal{M}}(\lambda) \leq \mathsf{negl}(\lambda)$.

A.1 Proof of Security for Section 7 Construction

We now prove Theorem 7.1.

Proof. This proof is similar to the one of the base scheme in Section 4.1. We discuss the differences here.

As before, the reduction algorithm receives an RSA challenge (N, e^*, h) and needs to use the attacker to compute $h^{1/e^*} \bmod N$. We extend the definition of a "conforming" attacker as one that will always make (1) a signing query for the forgery identity on the period t^* that it forges on, and (2) queries the random oracle on any identity that it uses in the forgery. Our proof will assume a conforming attacker.

Let $Q = Q(\lambda)$ be the maximum number of unique inputs that $\mathcal{A}$ queries to the random oracle G.

We describe the changes required to the sequence of games for the ID-based proof.

Game 1: (Security Game) This game is the same as the security game of the ID scheme.

Game 2: (Guessing the forgery period, identity index and part of its queried message) As in the base Game 2, the game guesses $t' \in [1, T]$, $\alpha \in [1, k]$ and $\beta \in \{0, 1\}^\ell$. It also guesses the index $\mu \in [1, Q]$ of the unique identity queried to G will be the identity $ID^* \in \mathcal{I}$ that is used as the forgery identity. An adversary wins this game iff: (1) it would have won in Game 1 with a forgery on period t^* for identity ID^* with message $M^* = m_1^*|m_2^*| \dots |m_k^*$ where some message $M = m_1|m_2| \dots |m_k$ was queried to the signing oracle for identity ID^* on period t^*, (2) $t' = t^*$, (3) $\beta = m_\alpha$ and (4) $m_\alpha \neq m_\alpha^*$.

Games 3-7: Unchanged.

Proof that the adversary's advantage in winning Game i and Game $i+1$ are within a polynomial follows from the claims in Section 4.1, except that the additional guess of the forgery identity in Game 2 reduces that relationship by an additional factor of $1/|\mathcal{I}|$. Formally, we have:

Claim A.3

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 2] \geq \frac{\mathbf{Adv}_{\mathcal{A}}[\text{Game } 1]}{T \cdot k \cdot 2^\ell \cdot Q}.$$

Proof. Since there is no change to the adversary's view of from Game 1, the probability of the adversary winning in Game 2 is the same as Game 1 times the probability of the game's guesses being correct. There is a $1/T$ probability of guessing the forging period, a $1/Q$ probability of guessing the forgery identity, at least a $1/k$ probability of guessing a message chunk in the signing query that will be different in the forgery (there may be more than one), and a 2^ℓ probability of guessing that chunk's value in the queried message. We note that this gives a polynomial-time reduction for whenever ℓ is polylogarithmic in λ. Recall that any adversary that is polynomial time in λ must give out a 1^T that is polynomially bounded in λ and can initiate at most a polynomial in λ number of random oracle queries. □

A.1.1 Main Reduction

We now show that if there exists a polynomial-time (in λ) attacker that has advantage $\epsilon = \epsilon(\lambda)$ in Game 7, then there exists a polynomial-time (in λ) attacker for the RSA problem in Assumption 3.1 with advantage ϵ.

On input an RSA challenge (N, e^*, h), the reduction algorithm proceeds as follows:

Setup.

1. Obtain $1^T, 1^n$ from the aggregate signature adversary $\mathcal{A}$.
2. Make random guesses of $t^* \in [1, T], \mu \in Q, \alpha \in [1, k], \beta \in \{0, 1\}^\ell, i^* \in [1, \lambda^3 + \lambda^2]$.
3. Choose a random PRF key K'. Let y' be the λ least significant bits of the RSA input e^* and set $c = y' \oplus F_{K'}(t^*, i^*)$. Choose a random prime $e_{\text{default}} \in [2^\lambda, 2^{\lambda+1} - 1]$. Set $K = (K', c, e_{\text{default}})$. Thus, note that by construction when i^* is the resolving index for t^*,

$$e_{t^*} = H_K(t^*) = 2^\lambda + (c \oplus F_{K'}(t^*, i^*)) = 2^\lambda + y' = e^*.$$

4. Send to $\mathcal{A}$ the public parameters $\mathsf{pp} = (T, N, K)$. To evaluate G, $\mathcal{A}$ must submit queries to the challenger.

G Queries. Proceeding adaptively, the adversary may request the evaluation of any $ID \in \mathcal{I}$. Let $Q = Q(\lambda)$ be the maximum number of unique inputs that $\mathcal{A}$ queries to the random oracle G or causes the challenger. If the input has been queried before, return that value. If this is the ith unique input, then choose random $u_0, \dots, u_k \in [0, N]$ and compute $Y = h^{\prod_{j=1}^{T} e_j}$. If $i = \mu$, then for $j = 1$ to k, where $j \neq \alpha$, set $U_j = Y^{u_j}$. Then set:

$$U_0 = (h^{-\beta})^{\prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_0} = h^{-\beta \prod_{i \neq t^*}^{T} e_i} \cdot h^{u_0 \prod_{i=1}^{T} e_i} \quad , \quad U_\alpha = h^{\prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_\alpha}.$$

Otherwise, for $j = 0$ to k, set $U_j = Y^{u_j}$.

Extract Queries. Proceeding adaptively, the adversary may request the private key for any $ID \in \mathcal{I}$ which was previously queried to G. (If not, the challenger instructs the adversary to do so.) If ID is the μ-th unique index queried to G, then abort. This corresponds to the same abort condition in Game 7. Otherwise, compute and return the private key $sk_{ID} = (d_0, \dots, d_k)$ as $d_i = h^{u_i}$, where the challenger chose these u_i values previously in a G query.

Signing Queries. For each period $t = 1$ to T, the adversary can request one signature on a message of its choice under any identities of its choice or skip that period. We assume the identity was previously queried to G; if not, it must do so. If the identity is not the μ-th unique query to G, then the challenger can compute the private key and execute the normal signing algorithm.

Otherwise, recall that the adversary must be conforming and thus will request some signature on the forgery period t^* for the forgery identity ID^*. However if $t = t^*$ and $m_\alpha = \beta$, it will abort as would Game 7. When asked to sign $M = m_1 | m_2 | \dots | m_k$ for period $t \neq t^*$, the reduction algorithm outputs:

$$\sigma = \left(h^{-\beta} \cdot h^{m_\alpha}\right)^{\prod_{i \neq t^*, i \neq t}^{T} e_i} \cdot \left(h^{u_0} \prod_{j=1}^{k} h^{u_j m_j}\right)^{\prod_{i \in T \setminus \{t\}} e_i} \tag{19}$$

$$= (h^{-\beta \prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_0})^{1/e_t} \cdot (\prod_{j=1, j \neq \alpha}^{k} U_j^{m_j})^{1/e_t} \cdot (h^{\prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_\alpha})^{m_\alpha / e_t} \tag{20}$$

$$= (U_0 \prod_{j=1}^{k} U_j^{m_j})^{1/e_t} \mod N. \tag{21}$$

and when $t = t^*$ and $m_\alpha = \beta$, it outputs the signature:

$$\sigma = \big(h^{u_0} \prod_{j=1}^{k} h^{u_j m_j}\big)^{\prod_{i \in T \setminus \{t\}} e_i} \tag{22}$$

$$= (1)^{\prod_{i \neq t^*, i \neq t}^{T} e_i} \cdot \big(h^{u_0} \prod_{j=1}^{k} h^{u_j m_j}\big)^{\prod_{i \in T \setminus \{t\}} e_i} \tag{23}$$

$$= (h^{-\beta} \cdot h^{m_\alpha})^{\prod_{i \neq t^*, i \neq t}^{T} e_i} \cdot \big(h^{u_0} \prod_{j=1}^{k} h^{u_j m_j}\big)^{\prod_{i \in T \setminus \{t\}} e_i} \tag{24}$$

$$= (h^{-\beta \prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_0})^{1/e_t} \cdot (\prod_{j=1, j \neq \alpha}^{k} U_j^{m_j})^{1/e_t} \cdot (h^{\prod_{i \neq t^*}^{T} e_i} \cdot Y^{u_\alpha})^{m_\alpha / e_t} \tag{25}$$

$$= (U_0 \prod_{j=1}^{k} U_j^{m_j})^{1/e_t} \mod N. \tag{26}$$

Output. Eventually $\mathcal{A}$ outputs a tuple $(t_f, (ID_1, \ldots, ID_z), (M_1, \ldots, M_z), \sigma_{agg})$. If the aggregate signature does not verify or if any of the reduction's guesses of $t^*, i^*, \mu, \alpha, \beta$ were incorrect, then abort. Note that these abort conditions are all consistent with the adversary not winning Game 7. We assume that the adversary has previously queried each identity in the list to G. (It would add at most z queries to Q to change the argument to cover this case.) We assume without loss of generality that $ID_1 = ID^*$, the μ-th unique identity queried to G. Let $E' = \prod_{i \in T/t^*} e_i$. We have:

$$\sigma_{agg}^{e^*} = \prod_{j=1}^{n} (U_{j,0} \prod_{i=1}^{k} U_{j,i}^{m_{j,i}}) \tag{27}$$

$$= (U_{1,0} \prod_{i=1}^{k} U_{1,i}^{m_{1,i}}) \cdot \prod_{j=2}^{n} (U_{j,0} \prod_{i=1}^{k} U_{j,i}^{m_{j,i}}) \tag{28}$$

$$= (h^{E'(\beta - m_\alpha)} \cdot Y^{u_0} \prod_{j=1}^{k} Y^{u_j m_j}) \cdot \prod_{j=2}^{n} (U_{j,0} \prod_{i=1}^{k} U_{j,i}^{m_{j,i}}) \tag{29}$$

Since the reduction can compute the e^*-th root of all values not in the h term, it can divide them out as:

$$\left(\frac{\sigma_{agg}}{\prod_{j=1}^{n} (g^{u_{j,0}} \prod_{i=1}^{k} g^{u_{j,i} m_{j,i}})^{E'}} \right)^{e^*} = \frac{(h^{E'(\beta - m_\alpha)} \cdot Y^{u_0} \prod_{j=1}^{k} Y^{u_j m_j}) \cdot \prod_{j=2}^{n} (U_{j,0} \prod_{i=1}^{k} U_{j,i}^{m_{j,i}})}{\prod_{j=1}^{n} (g^{u_{j,0}} \prod_{i=1}^{k} g^{u_{j,i} m_{j,i}})^{e^* \cdot E'}} \tag{30}$$

$$= h^{E'(\beta - m_\alpha)}. \tag{31}$$

Now, we have an equation of the form $x^a = y^b$, where $x = \frac{\sigma_{agg}}{\prod_{j=1}^{n} (g^{u_{j,0}} \prod_{i=1}^{k} g^{u_{j,i} m_{j,i}})^{E'}}$, $a = e^*$, $y = h$ and $b = E'(\beta - m_\alpha)$. Recall that the game would have already aborted if e^* was output for any period other than t^* and thus, $\gcd(e^*, E') = 1$. The game would also have aborted if $\beta = m_\alpha$. Finally since the $|\beta| = |m_\alpha| = \ell < \lambda$ and $e^* > 2^\lambda$, we can conclude that $\gcd(a, b) = 1$. This allows the reduction to apply Lemma 3.2 to efficiently compute $\hat{h} \in \mathbb{Z}_N$ such that $\hat{h}^{e^*} = h \mod N$. The reduction outputs this value as the RSA solution.

Analysis. We argue that the above reduction will succeed in outputting the RSA solution whenever the adversary wins in Game 7. The adversary's view in these scenarios differs only in the way that elements U_0 and U_α are chosen for the forgery identity by the random oracle G versus the challenger. They are chosen identically to the public keys of the base scheme in Game 7 versus the reduction. In both cases, they are chosen in a manner statistically close to choosing a random element in QR_N. The argument for this mirrors that of Section 4.1. It follows that the polynomial-time adversary cannot distinguish between these games.

$\square$