New Methods and Abstractions for RSA-Based Forward Secure Signatures

Susan Hohenberger[1,⋆] and Brent Waters[2,⋆⋆]

[1] Johns Hopkins University, susan@cs.jhu.edu
[2] University of Texas at Austin and NTT Research, bwaters@cs.utexas.edu

Abstract. We put forward a new abstraction for achieving forward-secure signatures that are (1) short, (2) have fast update and signing and (3) have small private key size. Prior work that achieved these parameters was pioneered by the pebbling techniques of Itkis and Reyzin (CRYPTO 2001) which showed a process for generating a sequence of roots $h^{1/e_1}, h^{1/e_2}, \ldots, h^{1/e_T}$ for a group element h in $\mathbb{Z}_N^*$. However, the current state of the art has limitations.

First, while many works claim that Itkis-Reyzin pebbling can be applied, it is seldom shown how this non-trivial step is concretely done. Second, setting up the pebbling data structure takes T time which makes key generation using this approach expensive (i.e., T time). Third, many past works require either random oracles and/or the Strong RSA assumption; we will work in the standard model under the RSA assumption.

We introduce a new abstraction that we call an *RSA sequencer*. Informally, the job of an RSA sequencer is to store roots of a public key U, so that at time period t, it can provide U^{1/e_t}, where the value e_t is an RSA exponent computed from a certain function. This separation allows us to focus on building a sequencer that efficiently stores such values, in a forward-secure manner and with better setup times than other comparable solutions. In addition, our sequencer abstraction has certain re-randomization properties that allow for constructing forward-secure signature schemes with a single trusted setup that takes T time and afterward individual key generation takes $\lg(T)$ time.

We demonstrate the utility of our abstraction by using it to provide concrete forward-secure signature schemes. We first give a random-oracle construction that closely matches the performance and structure of the Itkis-Reyzin scheme with the important exception that key generation can be realized much faster (after the one-time setup). We then move on to designing a standard model scheme. We believe this abstraction and illustration of how to use it will be useful for other future works.

We include a detailed performance evaluation of our constructions, with an emphasis on the time and space costs for large caps on the maximum number of time periods T supported. Our philosophy is that frequently updating forward secure keys should be part of "best practices" in key

⋆ Supported by NFS CNS-1414023, NSF CNS-1908181, the Office of Naval Research N00014-19-1-2294, and a Packard Foundation Subaward via UT Austin.

⋆⋆ Supported by NSF CNS-1414082, NSF CNS-1908611, Simons Investigator Award and Packard Foundation Fellowship.

maintenance. To make this practical, even for bounds as high as $T = 2^{32}$, we show that after an initial global setup, it takes only seconds to generate a key pair, and only milliseconds to update keys, sign messages and verify signatures. The space requirements for the public parameters and private keys are also a modest number of kilobytes, with signatures being a single element in $\mathbb{Z}_N$ and one smaller value.

1 Introduction

Compromise of cryptographic key material can be extremely costly for an organization to weather. In March of 2011 an attack on EMC allowed attackers to gain the master seeds for EMC's SecureID product. The compromise eventually led the company to offer replacements for the 40 million tokens at an estimated cost of $66 million USD [12]. Also in 2011, the certificate authority DigiNotar was compromised and found that several rogue certificates for companies such as Google were issued in Iran [10]. The attack led to DigiNotar's root certificate being removed from all major web browers. Eventually the firm filed for bankruptcy and cost its parent company, VASCO, millions of dollars [10].

One bulwark to mitigate the impact of private key compromise is the concept of forward security, which abstractly is meant to protect past uses of the private key material before a compromise by periodically updating or evolving the private key. In this work, we focus on the concrete case of forward secure signatures [3, 4]. In forward secure signatures, public keys are fixed but signatures that verify under this key can be generated by a private key associated with a period t. At any point, the private key holder can choose to evolve or update the private key to the next period $t + 1$.[3] After an update, the signing key is capable of creating signatures associated with period $t+1$, but *not* for any earlier period. Importantly, if an attacker compromises a private key at period t', it will be unable to forge signatures on any earlier period. Returning to the example of DigiNotar, if forward signatures were deployed (and assuming one could make a conservative estimate on the time of attack) the browsers could have revoked the root certificate starting at the time of compromise, but at least temporarily accepted earlier signatures, which would have allowed the organizations certified by DigiNotar more time to migrate to a new authority.

Since the introduction of forward secure signatures by Anderson [3] and Bellare and Miner [4], there have been several forward secure signature systems put forth in the literature. One can bifurcate solutions into two types. Those that are built from general signatures that follow a "tree-based" structure in which the depth of the tree and signature size grows logarithmically with the number of time periods T. And a second category of "hash-and-sign" signatures built in specific number theoretic contexts such as the RSA setting or in bilinear groups. The main appeal of the latter category is efficiency and that will be our focus.

[3] Key updates could correspond to actual time intervals or be done in some other arbitrary manner.

In this second category the work of Itkis and Reyzin [22] (pebbling variant) is notable for giving the first "hash-and-sign" scheme (using the random oracle model) with fast signing and key update and small ($\lg(T)$ sized) private keys. They do this by introducing a novel "pebbling" technique that allows the signer to compute successive roots $h^{1/e_1}, h^{1/e_2}, \ldots, h^{1/e_T}$ of a group element h (mod N). This technique was used in many other works including Camenisch and Koprowski [9] which use it to achieve standard-model forward-secure signatures with similar parameters to Itkis-Reyzin under the Strong RSA Assumption.

There are three limitations, however, with the current state of the art in pebbling solutions. First, most subsequent works (e.g., [22, 9, 1]) that claim to apply Itkis-Reyzin pebbling simply state that Itkis-Reyzin pebbling applies, but do not concretely show how to do this. This creates a critical technical gap where there is an intuitive understanding of what the pebbling version of the forward-secure scheme is, but no precise description of that scheme (and in our experience working out these details is non-trivial). The issue appears to arise from the fact that the original Itkis-Reyzin pebbling techniques are not abstracted and defined out as a primitive that can be immediately reused in other works. The second limitation is that these pebbling techniques require the setup time for each scheme to be linear in T which can be prohibitive. The third limitation is that some solutions require the Strong RSA Assumption.

We address all of these issues with an abstraction called an RSA-sequencer. Intuitively, this sequencer performs a function commensurate with earlier pebbling work, but abstracted in a way that allows it to be readily applied for proving schemes in a formal manner. In addition, our sequencer allows for a single global setup that will run in time T to produce a data structure of size $\lg(T)$ group elements. Subsequently, the output of the global setup can be re-randomized in a way that allows for forward secure signatures with fast ($\lg(T)$ operations) key generation. Using our abstraction we are able to obtain concretely defined hash-and-sign forward secure signatures in both the standard and random oracle model. We then give concrete performance evaluations of these.

RSA Sequencers. We introduce an RSA Sequencer concept comprised of five *deterministic* algorithms ($\mathsf{SeqSetup}$, $\mathsf{SeqUpdate}$, $\mathsf{SeqCurrent}$, $\mathsf{SeqShift}$, $\mathsf{SeqProgram}$). We begin with an informal overview here. Section 4 contains a formal description.

Let N be an RSA modulus and H be a function from $[1, T]$ to positive integers where we'll use the notation $e_i = H(i)$. In addition, consider a tuple $(v_1, \ldots, v_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}$. For each j, let $V_j = v_j^{\prod_{i \in [1,T]} e_i}$. Intuitively, the purpose of the sequencer is when it is at period t to be able to output $V_1^{1/e_t}, \ldots, V_{\mathtt{len}}^{1/e_t}$.[4]

A call to $\mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}))$ will produce a "state" output that we denote $\mathtt{state}_1$. Next, if we call $\mathsf{SeqUpdate}(\mathtt{state}_1)$ we get another state $\mathtt{state}_2$. The update algorithm can be repeated iteratively to compute

[4] For the purposes of this overview, we will implicitly assume that all e_i values are relatively prime to $\phi(N)$ and thus V_j^{1/e_i} is uniquely defined. However, this is not required in our formal specification.

$\mathtt{state}_t$ for any $t \in [1, T]$. Finally, a call to $\mathsf{SeqCurrent}(\mathtt{state}_t)$ will give as output $V_1^{1/e_t}, \ldots, V_{\mathtt{len}}^{1/e_t}$. These three algorithms together form the core functionality. We now turn to the last two.

Consider a set of integers (i.e., exponents) $z_1, \ldots, z_{\mathtt{len}}$ along with group elements $g_1, \ldots, g_{\mathtt{len}} \in \mathbb{Z}_N^*$ where we let $v_1 = g_1^{z_1}, \ldots, v_{\mathtt{len}} = g_{\mathtt{len}}^{z_{\mathtt{len}}}$. Then it is the case that a call to $\mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (g_1, \ldots, g_{\mathtt{len}}))$ that produces $\mathtt{state}'$ followed by a call to $\mathsf{SeqShift}(\mathtt{state}', (z_1, \ldots, z_{\mathtt{len}}))$ produces the same output as a call to $\mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}))$.

Why would one want such a functionality? At first it seems superfluous as one can reach the same endpoint without bothering with the SeqShift algorithm. Looking forward in our RSA Sequencer construction the SeqShift will be a significantly cheaper function to call as its computation time will scale proportionally to $\lg(T)$, while the SeqSetup algorithm will run in time proportional to T. In the schemes we build, we can save computation costs by letting a trusted party pay a one time cost of running SeqSetup to generate a set of global parameters. Then with these parameters, each individual party will be able to generate their public/private keys much more cheaply using the SeqShift algorithm.

Finally, we arrive at the SeqProgram algorithm. This algorithm will actually not be used in our constructions proper, but instead be used by the reduction algorithm to generate a compromised key in the proof of forward security. Thus the performance of this algorithm is less important, other than it must run in polynomial time. For any value $\mathtt{start} \in [1, T]$, consider a tuple $v'_1 = v_1^{\prod_{i \in [1, \mathtt{start}-1]} e_i}, \ldots, v'_{\mathtt{len}} = v_{\mathtt{len}}^{\prod_{i \in [1, \mathtt{start}-1]} e_i}$. Then $\mathsf{SeqProgram}(N, 1^T, H, 1^{\mathtt{len}}, (v'_1, \ldots, v'_{\mathtt{len}}), \mathtt{start})$ produces the same output as $\mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}))$ followed by $\mathtt{start} - 1$ iterative calls to SeqUpdate. Intuitively, the semantics of SeqProgram provide an interface to generate the $\mathtt{start}$-th private key without knowing any of the first $\mathtt{start} - 1$ roots of $V_1, \ldots, V_{\mathtt{len}}$.

An important point we wish to emphasize is that the RSA Sequencer definitions we give only have correctness properties and do not contain any security definitions. Issues like choosing a proper RSA modulus N and a hash function H are actually outside the RSA Sequencer definition proper and belong as part of the cryptosystems building on top of them.

In Section 5, we provide an efficient RSA Sequencer. The construction itself is closely adapted from a key storage mechanism by Hohenberger and Waters [14] used for synchronized aggregate signatures that could support T synchronization periods with $\lg(T)$ private key storage. This storage mechanism in turn had conceptual roots in the pebbling optimization by Itkis and Reyzin [15] for forward secure signatures. The RSA Sequencer bears some history and resemblance to accumulators [6], but has different goals, algorithms and constructions.

In our construction the (optimized version of the) SeqSetup algorithm makes T calls to H and performs $T \cdot \mathtt{len}$ exponentiations. If we break the abstraction slightly and let a trusted party running it know $\phi(N)$ the exponentiations can be replaced with T multiplications mod $\phi(N)$ and $2 \cdot \mathtt{len}$ exponentiations. The space overhead of the states (which will translate to private key size) will be at most $2\lg(T)$ elements of $\mathbb{Z}_N^*$. The SeqUpdate algorithm will invoke at most $\lg(T)$

calls to H and $\lg(T) \cdot \texttt{len}$ exponentiations. The SeqShift algorithm will invoke at most $2 \cdot \lg(T) \cdot \texttt{len}$ exponentiations and no calls to H. Finally, the call to SeqCurrent is simply a lookup and thus essentially of no cost.

Building Forward-Secure Signatures with the RSA Sequencer Our work illustrates the value of the RSA Sequencer abstraction by showing how to use it. We show this concretely in Section 6 (random oracle model) and Section 7 (standard model). For space reasons, our detailed intuition on how we do this is deferred to the full version. In a nutshell, however, when instantiated with our logarithmic-update sequencer construction of Section 5, for $T = 2^{32}$, we obtain forward-secure schemes with key generations in the milliseconds (random oracle) or seconds (standard model), whereas pebbling Itkis-Reyzin [15] (or Camenisch-Koprowski [9]) takes 20 days and tree-based MMM [18] takes roughly 7.8 years! We provide further performance comparisons in the full version.

In our standard-model construction, we are limited to giving out one signature per key update. Or put another way the signer must execute a key update operation after every signature. Arguably, this should actually be considered to be the "best possible" key hygiene in the sense that we get forward security on a per signature granularity basis. In the event that the user accidentally issues more than one signature during time period t, the forward security property guarantees that all signatures issued before t remain secure. Moreover, as we discuss in Section 7, for our particular construction, all signatures issued after t appear to remain secure as well.

If this single-sign restriction is considered too burdensome, it can be removed using an idea common in the literature where the forward-secure scheme is combined with a regular signature scheme. During each update, the signer generates a temporary public/private key pair for a standard (not forward-secure) signature scheme. She then uses the forward-secure signing algorithm to sign a certificate for this new (temporary) public key. Now all signatures in this period are first signed with the temporary private key and the final signature consists of this signature along with the attached temporary public key and its certificate.

Our constructions make non-black box use of the RSA Sequencer, however, investigating more general methods and sequencers that could be used in a black-box manner is an interesting open problem.

1.1 Further Related Work Discussion

Krawczyk [17] provided a generic construction from any signature scheme where the the public key and signatures have size independent of T, but the signer's storage grows linearly with T. Abdalla and Reyzin [2] showed how to shorten the private keys in the "hash-and-sign" Bellare-Miner [4] construction in the random oracle model. Itkis and Reyzin [15] presented GQ-based signatures with "optimal" signing and verification in the random oracle model using a very elegant pebbling approach. Camenisch and Koprowski [9] use it to achieve standard-model forward-secure signatures with similar parameters under the Strong RSA Assumption. Our later constructions will have mechanics and performance close

to these schemes, with the exceptions that we offer much faster key generation times and require only the (regular) RSA Assumption.

Kozlow and Reyzin [16] presented the KREUS construction that allows for very fast key update at the cost of longer signing and verification times. We observe that one can derive a weakly secure one-time signature secure from the RSA assumption by combining the RSA Chameleon Hash function of Bellare and Ristov [5] with a transformation due to Mohassel [19]. If we consider our Section 7 scheme with a single message chunk (i.e. $k = 1$) and the randomness terms for full security stripped away, then the signatures produced at each time period correspond to this signature scheme.

2 Definitions

Following prior works [4, 15], we begin with a formal specification for a *key-evolving signature* and then capture the security guarantees we want from such a scheme in a *forward-security* definition. Informally, in a key-evolving signature, the key pair is created to consist of a (fixed) public key and an initial secret key for time period 1. This secret key can then be locally updated by the key holder up to a maximum of T times. Crucial to security, the signer must delete the old secret key sk_t after the new one sk_{t+1} is generated. Any signature produced with the initial or any one of the updated secret keys will verify with respect to the fixed public key pk. Our specification below follows Bellare and Miner [4] with the exception that we introduce a global setup algorithm. Our specification can be reduced to theirs by having each signer run its own setup as part of the key generation algorithm. However, as we will later see in our constructions, some significant efficiency improvements can be realized by separating out and "re-using" a set of public parameters.

Definition 1 (Key-Evolving Signatures [4, 15]). *A key-evolving signature scheme for a max number of periods T and message space $\mathcal{M}(\cdot)$ is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ *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* $(\mathsf{pk}, \mathsf{sk}_1)$*. Notationally, we will assume that the time period of the key can be easily extracted from the secret key.*

Update$(\mathsf{pp}, \mathsf{sk}_t)$ *: On input the public parameters* pp*, the update algorithm takes in a secret key sk_t for the current period $t \leq T$ and returns the secret key sk_{t+1} for the next period $t+1$. By convention, we set that sk_{T+1} is the empty string and that* $\mathsf{Update}(\mathsf{pp}, \mathsf{sk}_T, T)$ *returns* sk_{T+1}.

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

Verify$(\mathsf{pp}, \mathsf{pk}, m, t, \sigma)$ *: On input the public parameters* pp*, the verification algorithm takes in a public key* pk*, a message $m \in \mathcal{M}(\lambda)$, a period $t \leq T$ and*

a purported signature σ, and returns 1 if and only if the signature is valid and 0 otherwise.

Correctness. Let $\mathsf{poly}(x)$ *denote the set of polynomials in* x*. For a key-evolving scheme, the correctness requirement stipulates that for all* $\lambda \in \mathbb{N}$*,* $T \in \mathsf{poly}(\lambda)$*,* $\mathsf{pp} \in \mathsf{Setup}(1^\lambda, 1^T)$*,* $(\mathsf{pk}, \mathsf{sk}_1) \in \mathsf{KeyGen}(\mathsf{pp})$*,* $1 \leq t \leq T$*,* $m \in \mathcal{M}(\lambda)$*,* $\mathsf{sk}_{i+1} \in \mathsf{Update}(\mathsf{pp}, \mathsf{sk}_i)$ *for* $i = 1$ *to* T*,* $\sigma \in \mathsf{Sign}(\mathsf{pp}, \mathsf{sk}_t, m)$*, it holds that*

$$\mathsf{Verify}(\mathsf{pp}, \mathsf{pk}, m, t, \sigma) = 1.$$

We now turn to capturing the forward-security guarantee desired, which was first formalized by Bellare and Miner [4] and in turn built on the Goldwasser, Micali and Rivest [11] security definition for digital signatures of unforgeability with respect to adaptive chosen-message attacks. Intuitively, in the foward-security game, the adversary will additionally be given the power to "break in" to the signer's computer and capture her signing key sk_b at any period $1 < b \leq T$. The adversary's challenge is to produce a valid forgery for any time period $j < b \leq T$.

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

Setup: The adversary sends 1^T to the challenger, who runs $\mathsf{Setup}(1^\lambda, 1^T)$ to obtain the public parameters pp.[5] Then the challenger runs $\mathsf{KeyGen}(\mathsf{pp})$ to obtain the key pair $(\mathsf{pk}, \mathsf{sk}_1)$. The adversary is sent $(\mathsf{pp}, \mathsf{pk})$.

Queries: From $t = 1$ to T, the challenger computes sk_{t+1} via $\mathsf{Update}(\mathsf{pp}, \mathsf{sk}_t)$. If the adversary issues a signing query for message $m \in \mathcal{M}$ for time period $1 \leq t \leq T$, then the challenger responds with $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}_t, m)$ and puts (m, t) in a set C. When the adversary issues her break-in query for period $1 < b \leq T$, the challenger responds with sk_b.[6] If the adversary does not choose to make a break-in query, then set $b = T + 1$.

Output: Eventually, the adversary outputs a tuple (m, t, σ) and wins the game if:

1. $1 \leq t < b$ (i.e., before the break-in); and
2. $m \in \mathcal{M}$; and
3. $(m, t) \notin C$; and
4. $\mathsf{Verify}(\mathsf{pp}, \mathsf{pk}, m, t, \sigma) = 1$.

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.

[5] Any adversary $\mathcal{A}$ that runs in time polynomial in λ will be restricted (by its own running time) to responding with a T value that is polynomial in λ.

[6] Technically, it is non-limiting to allow the adversary only one break-in period, because from this secret key she can run the update algorithm to produce valid signing keys for all future periods. Her forgery must, in any event, come from a period prior to her earliest break-in.

Definition 2 (Forward Security). *A key-evolving signature scheme Π for message space $\mathcal{M}$ is forward secure if for all probabilistic polynomial-time in λ adversaries $\mathcal{A}$, there exists a negligible function* negl*, such that* $\mathsf{SigAdv}_{\mathcal{A},\Pi,\mathcal{M}}(\lambda) \leq \mathsf{negl}(\lambda)$.

Single Sign. In the above definition, the adversary can request multiple signatures for each time period. We will also be considering schemes where an honest signer is required to update his secret key after each signature, and thus the adversary will be restricted to requesting at most one message signed per period. Formally, during Queries, the challenger will only respond to a signing request on (m, t) if $m \in \mathcal{M}$, $1 \leq t \leq T$, and there is no pair of the form $(x, t) \in C$. We will call schemes with this restriction *single sign* key-evolving schemes and the corresponding unforgeability notion will be called *single sign* forward security.

Weakly Secure. For any signature scheme, one can also consider a variant of the security game called *existential unforgeability with respect to weak chosen-message attacks (or weakly secure)* (e.g., see Boneh and Boyen [7]) where, at the beginning of the security game, the adversary must send to the challenger a set Q of the messages that she will request signatures on. In the case of forward security, Q must contain the message-period pairs (m_i, t_i). Instead of making any adaptive signing queries, the challenger will simply produce signatures on all of these messages for their corresponding period. Then the adversary must produce a forgery for some $(m^*, t^*) \notin Q$.

3 Number Theoretic Assumptions

We use the variant of the RSA assumption [20] involving safe primes. A *safe prime* is a prime number of the form $2p + 1$, where p is also a prime.

Assumption 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 $p'q'$. Choose $x \in \mathrm{QR}_N$ and compute $h = x^e \mod N$. Given (N, e, h), it is hard to compute x such that $h = x^e \mod N$.*

4 RSA Sequencers

Shortly, we will present forward-secure signature constructions in the RSA setting. All of these constructions and their proofs make use of an abstraction we call an *RSA Sequencer*. We now provide a specification for this abstraction, as well as minimum efficiency and correctness requirements. In Section 5, we provide an efficient construction.

Definition 3 (RSA Sequencer). *An RSA Sequencer consists of a tuple of* deterministic *algorithms* (SeqSetup, SeqUpdate, SeqCurrent, SeqShift, SeqProgram) *such that:*

SeqSetup($N \in \mathbb{Z}, 1^T, H : \{1, \ldots, T\} \to \mathbb{Z}, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}$) : On input of a positive integer N, the number of time periods T, a function H from $[1, T]$ to positive integers, a positive integer `len` and a `len`-tuple of elements in $\mathbb{Z}_N$, the SeqSetup algorithm outputs a state value `state`.

SeqUpdate(`state`) : On input of a state value `state`, the SeqUpdate algorithm produces another value `state`$'$.

SeqCurrent(`state`) : On input of a state value `state`, the SeqCurrent algorithm produces a tuple $(s_1, \ldots, s_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}$.

SeqShift(`state`, $(z_1, \ldots, z_{\mathtt{len}}) \in \mathbb{Z}^{\mathtt{len}}$) : On input of a state value `state` and a `len`-tuple of integers, the SeqShift algorithm produces another value `state`$'$.

SeqProgram ($N \in \mathbb{Z}, 1^T, H : \{1 \ldots, T\} \to \mathbb{Z}, 1^{\mathtt{len}}, (v'_1, \ldots, v'_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}, \mathtt{start} \in \{1, \ldots, T\}$): On input of a positive integer N, the number of time periods T, a function H from $[1, T]$ to positive integers, a positive integer `len`, a `len`-tuple of elements in $\mathbb{Z}_N$ and an integer `start` $\in [1, T]$, the SeqProgram algorithm outputs a state value `state`.

We note that the SeqProgram algorithm will not appear in our signature constructions, but instead be employed solely in the proof of forward security.

(Minimum) Efficiency We require that the SeqSetup and SeqProgram algorithms run in time polynomial in their respective inputs and all other algorithms run in time polynomial in $\lg(N), T$ and `len` and the time to evaluate H.

Correctness We specify three correctness properties of an RSA Sequencer. Our specification implicitly relies on the fact that all of the algorithms (including SeqSetup) are deterministic. We also use the shorthand that $e_t = H(t)$ for $t \in [1, T]$. The correctness properties are:

Update/Output Correctness For any $N \in \mathbb{Z}, T \in \mathbb{Z}, H : \{1 \ldots, T\} \to \mathbb{Z}, \mathtt{len} \in \mathbb{Z}, (v_1, \ldots, v_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}$, the following must hold: Let $\mathtt{state}_1 = \mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}))$. For $t = 2$ to T, let $\mathtt{state}_t = \mathsf{SeqUpdate}(\mathtt{state}_{t-1})$. Then for all $t \in [1, T]$, it must be that

$$\mathsf{SeqCurrent}(\mathtt{state}_t) = (v_1^{\prod_{i \in [1,T] \setminus \{t\}} e_i}, \ldots, v_{\mathtt{len}}^{\prod_{i \in [1,T] \setminus \{t\}} e_i})$$

where the arithmetic is done in $\mathbb{Z}_N$.

Shift Correctness For any $N \in \mathbb{Z}, T \in \mathbb{Z}, H : \{1 \ldots, T\} \to \mathbb{Z}, \mathtt{len} \in \mathbb{Z}, (v_1, \ldots, v_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}$ and $(z_1, \ldots, z_{\mathtt{len}}) \in \mathbb{Z}^{\mathtt{len}}$, the following must hold: Let $\mathtt{state} = \mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}))$. Let $v'_1 = v_1^{z_1}, \ldots, v'_{\mathtt{len}} = v_{\mathtt{len}}^{z_{\mathtt{len}}}$ (all in $\mathbb{Z}_N$) and $\mathtt{state}' = \mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v'_1, \ldots, v'_{\mathtt{len}}))$, then it must hold that

$$\mathtt{state}' = \mathsf{SeqShift}(\mathtt{state}, (z_1, \ldots, z_{\mathtt{len}})).$$

One could define a stronger form of shift correctness that holds after any number of updates; however, we will only need this to hold for when SeqShift is operated immediately on the initial state output of SeqSetup.

Program Correctness For any $N \in \mathbb{Z}, T \in \mathbb{Z}, H : \{1 \ldots, T\} \to \mathbb{Z}, \mathtt{len} \in \mathbb{Z}, (v_1, \ldots, v_{\mathtt{len}}) \in \mathbb{Z}_N^{\mathtt{len}}, \mathtt{start} \in [1, T+1]$, the following must hold: Let $\mathtt{state}_1 = \mathsf{SeqSetup}(N, 1^T, H, 1^{\mathtt{len}}, (v_1, \ldots, v_{\mathtt{len}}))$. For $t = 2$ to $\mathtt{start}$, let $\mathtt{state}_t = \mathsf{SeqUpdate}(\mathtt{state}_{t-1})$. Let $v'_1 = v_1^{\prod_{i \in [1, \mathtt{start}-1]} e_i}, \ldots v'_{\mathtt{len}} = v_{\mathtt{len}}^{\prod_{i \in [1, \mathtt{start}-1]} e_i}$ (all in $\mathbb{Z}_N$). Finally let $\mathtt{state}' = \mathsf{SeqProgram}(N, 1^T, H, 1^{\mathtt{len}}, (v'_1, \ldots, v'_{\mathtt{len}}), \mathtt{start})$. It must hold that $\mathtt{state}_{\mathtt{start}} = \mathtt{state}'$.

5 Our Sequencer Construction

We now give an RSA sequencer construction where the number of hashes and exponentiations for update is logarithmic in T. Furthermore, the storage will consist of a logarithmic in T number of elements of $\mathbb{Z}_N$. Our sequencer construction will follow closely in description to the key storage technique from Hohenberger and Waters [14] and is also conceptually similar to the pebbling optimization from Itkis and Reyzin [15].

Let's recall the purpose of an RSA sequencer. Let N be an integer that we'll think of as an RSA modulus and H be a function from $[1, T]$ to positive integers where we'll use the notation $e_i = H(i)$. Focusing on the length $\mathtt{len} = 1$ case, a sequencer will be given as input a value $v \in \mathbb{Z}_N$ and we let $V = v^{\prod_{i \in [1,T]} e_i}$.

The goal of a sequencer is two fold. First, after k calls to SeqUpdate, the SeqCurrent call should output $V^{1/e_{k+1}}$. Second, it should be the case that it has a forward security property where one cannot compute $V^{1/e_{k'+1}}$ for $k' < k+1$ from the data structure. One easy way to achieve these goals is that after k calls to SeqUpdate the data structure can simply store $v^{\prod_{i \in [1,k]} e_i}$. In this manner the SeqUpdate algorithm only needs a single exponentiation to update the data structure, but the SeqCurrent algorithm will need $T - k - 1$ exponentiations to compute $V^{1/e_{k+1}}$ from $v^{\prod_{i \in [1,k]} e_i}$.

Instead we use a more complex data structure that stores logarithmic in T "partial computations". After k calls to SeqUpdate, the data structure will already have $V^{1/e_{k+1}}$ ready for retrieval. Moreover, the next SeqUpdate call will do a logarithmic amount of work that has the next one ready as well. Intuitively, each call to SeqUpdate will perform work that both applies to computing "nearby" roots as well as progress towards further out time periods. The description below gives the details and supports a tuple of length $\mathtt{len}$.

For ease of exposition, we will assume that the setup algorithm only accepts values of T for which there is an integer $\mathtt{levels}$ where $T = 2^{\mathtt{levels}+1} - 2$. The storage will consist of an integer $\mathtt{index}$ that determines the current period and a sequence of sets $S_1, \ldots, S_{\mathtt{levels}}$ storing "partial computations" where elements of set S_i are of the form

$$(w_1, \ldots, w_{\mathtt{len}}) \in \mathbb{Z}_N^{*\mathtt{len}}, \mathtt{open} \in [1, T], \mathtt{closing} \in [1, T], \mathtt{count} \in [1, T].$$

Here if R is the set of integers $[\texttt{open}, \texttt{open}+2^{i-1}-1] \cup [\texttt{closing}+\texttt{count}, \texttt{closing}+2^{i-1}-1]$, then $w_i = v_i^{\prod_{j \in [1,T] \setminus R} e_j}$. Here and throughout this work, we use as shorthand $e_j = H(j)$. We begin with giving the descriptions of and proving correctness of all of the algorithms except the SeqProgram algorithm which we will circle back to at the end of the section.

SeqSetup$(N, 1^T, H, 1^{\texttt{len}}, (v_1, \ldots, v_{\texttt{len}}))$ Initialize sets $S_1, \ldots, S_{\texttt{levels}}$ to be empty. Then for $i = 2$ to levels perform the following:

- Let $R = [2^i - 1, 2^{i+1} - 2]$.
- Compute $w_1 = v_1^{\prod_{j \in [1,T] \setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j \in [1,T] \setminus R} e_j}$.
- Put in S_i $((w_1^{e_{(2^i-1)+2^{i-1}}}, \ldots, w_{\texttt{len}}^{e_{(2^i-1)+2^{i-1}}}), 2^i - 1, (2^i - 1) + 2^{i-1}, 1)$.
- Put in S_i $((w_1, \ldots, w_{\texttt{len}}), (2^i - 1) + 2^{i-1}, 2^i - 1, 0)$.

Finally, let $R = [1, 2]$ and compute $w_1 = v_1^{\prod_{j \in [1,T] \setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j \in [1,T] \setminus R} e_j}$. Put in S_1 $((w_1, \ldots, w_{\texttt{len}}), 2, 1, 0)$. And set $\texttt{current} = (w_1^{e_2}, \ldots, w_{\texttt{len}}^{e_2})$.

The output is $\texttt{state} = \big(\texttt{index} = 1, \texttt{current}, (S_1, \ldots, S_{\texttt{levels}})\big)$.

SeqUpdate(state) For $i = 1$ to levels, perform the following:

- Find a tuple (if any exist) in S_i of $((w_1, \ldots, w_{\texttt{len}}), \texttt{open}, \texttt{closing}, \texttt{count})$ with the smallest open value.[7]
- Replace it with a new tuple $((w'_1 = w_1^{e_{\texttt{closing}+\texttt{count}}}, \ldots, w'_{\texttt{len}} = w_{\texttt{len}}^{e_{\texttt{closing}+\texttt{count}}}), \texttt{open}' = \texttt{open}, \texttt{closing}' = \texttt{closing}, \texttt{count}' = \texttt{count}+1)$ where $((w'_1, \ldots, w'_{\texttt{len}}), \texttt{open}', \texttt{closing}', \texttt{count}')$ is the newly added tuple.

Then for $i =$ levels down to 2,

- Find a tuple (if any) of the form $((w_1, \ldots, w_{\texttt{len}}), \texttt{open}, \texttt{closing}, \texttt{count} = 2^{i-1})$ in S_i.
- Remove this tuple from the set S_i.
- To the set S_{i-1}, add the tuple $((w'_1 = w_1, \ldots, w'_{\texttt{len}} = w_{\texttt{len}}), \texttt{open}' = \texttt{open}, \texttt{closing}' = \texttt{open} + 2^{i-2}, \texttt{count}' = 0)$ where $((w'_1, \ldots, w'_{\texttt{len}}), \texttt{open}', \texttt{closing}', \texttt{count}')$ is the newly added tuple.
- Also add to the set S_{i-1}, the tuple $((w'_1 = w_1, \ldots, w'_{\texttt{len}} = w_{\texttt{len}}), \texttt{open}' = \texttt{open} + 2^{i-2}, \texttt{closing}' = \texttt{open}, \texttt{count}' = 0)$.

Finally, from S_1 find the tuple $((w_1, \ldots, w_{\texttt{len}}), \texttt{open} = \texttt{index}+1, \texttt{closing}, 1)$. Remove this from S_1. Set $\texttt{index}' = \texttt{index} + 1$ and $\texttt{current}' = (w_1, \ldots, w_{\texttt{len}})$. The output is $\texttt{state}' = \big(\texttt{index}', \texttt{current}', (S_1, \ldots, S_{\texttt{levels}})\big)$.

SeqCurrent(state) On input $\texttt{state} = \big(\texttt{index}, \texttt{current}, (S_1, \ldots, S_{\texttt{levels}})\big)$, the algorithm simply outputs $\texttt{current} = (w_1, \ldots, w_{\texttt{len}})$.

[7] In a particular S_i there might be zero, one or two tuples. If there are two, the one with the larger open value is ignored. Ties will not occur, as our analysis will show.

$\mathsf{SeqShift}(\mathtt{state}, (z_1, \ldots, z_{\mathtt{len}}))$ For $i = 1$ to $\mathtt{levels}$, find each tuple (if any exist) in S_i of the form $((w_1, \ldots, w_{\mathtt{len}}), \mathtt{open}, \mathtt{closing}, \mathtt{count})$. Then replace it with a new tuple $((w'_1 = w_1^{z_1}, \ldots, w'_{\mathtt{len}} = w_{\mathtt{len}}^{z_{\mathtt{len}}}), \mathtt{open}' = \mathtt{open}, \mathtt{closing}' = \mathtt{closing}, \mathtt{count}' = \mathtt{count})$. Finally, set $\mathtt{current}' = (w_1^{z_1}, \ldots, w_{\mathtt{len}}^{z_{\mathtt{len}}})$. The output is $\mathtt{state}' = \big(\mathtt{index}, \mathtt{current}', (S_1, \ldots, S_{\mathtt{levels}})\big)$.

We discuss the efficiency and correctness of the above in the full version.

5.1 The SeqProgram Algorithm

We conclude with describing the SeqProgram algorithm. Intuitively, at many places we are required to compute $v^{\prod_{j \in [1,T] \setminus R} e_j}$ for some set of values R. That is we need to raise v to all e_j values except those in the set R. However, instead of being given v the algorithm is given $v' = v^{\prod_{i \in [1, \mathtt{start}-1]} e_i}$. Therefore we must check (in the correctness argument) that in every case $R \cap [1, \mathtt{start} - 1] = \emptyset$. If so, we can let $X = [1, T] \setminus (R \cup [1, \mathtt{start} - 1])$ and compute $(v')^{\prod_{j \in X} e_j} = v^{\prod_{j \in [1,T] \setminus R} e_j}$.

$\mathsf{SeqProgram}(N, 1^T, H, 1^{\mathtt{len}}, (v'_1, \ldots, v'_{\mathtt{len}}), \mathtt{start})$ The algorithm first sets the value $\mathtt{index} = \mathtt{start}$. Next for each $i \in [1, \mathtt{levels}]$ the algorithm inserts tuples according to the following description.

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

Case 2: Not Case 1 and $\mathtt{index} = k \cdot 2^i + r$ for $0 \leq r < 2^{i-1}$. The algorithm will place two elements in S_i. First, let $\mathtt{open} = (k+1) \cdot 2^i - 1, \mathtt{closing} = (k+1) \cdot 2^i - 1 + 2^{i-1}$ and $\mathtt{count} = r$. Then let $R = [\mathtt{open}, \mathtt{open} + 2^{i-1} - 1] \cup [\mathtt{closing} + \mathtt{count}, \mathtt{closing} + 2^{i-1} - 1]$ and let $X = [1, T] \setminus (R \cup [1, \mathtt{index} - 1])$. The first one it places is

$$((w_1 = (v'_1)^{\prod_{j \in X} e_j}, \ldots, w_{\mathtt{len}} = (v'_{\mathtt{len}})^{\prod_{j \in X} e_j}), \mathtt{open}, \mathtt{closing}, \mathtt{count}).$$

To create the second tuple, let $\mathtt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}$, $\mathtt{closing} = (k+1) \cdot 2^i - 1$ and $\mathtt{count} = 0$. Next let $R = [\mathtt{open}, \mathtt{open} + 2^{i-1} - 1] \cup [\mathtt{closing} + \mathtt{count}, \mathtt{closing} + 2^{i-1} - 1]$ and let $X = [1, T] \setminus (R \cup [1, \mathtt{index} - 1])$.

$$((w_1 = (v'_1)^{\prod_{j \in X} e_j}, \ldots, w_{\mathtt{len}} = (v'_{\mathtt{len}})^{\prod_{j \in X} e_j}), \mathtt{open}, \mathtt{closing}, \mathtt{count}).$$

Case 3: Not Case 1 and $\mathtt{index} = k \cdot 2^i + r$ for $2^{i-1} \leq r < 2^i$. The algorithm inserts a single element. First, let $\mathtt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}$, $\mathtt{closing} = (k+1) \cdot 2^i - 1$ and $\mathtt{count} = r - 2^{i-1}$. Then let $R = [\mathtt{open}, \mathtt{open} + 2^{i-1}] \cup [\mathtt{closing} + \mathtt{count}, \mathtt{closing} + 2^{i-1}]$ and let $X = [1, T] \setminus (R \cup [1, \mathtt{index} - 1])$.

$$((w_1 = (v'_1)^{\prod_{j \in X} e_j}, \ldots, w_{\mathtt{len}} = (v'_{\mathtt{len}})^{\prod_{j \in X} e_j}), \mathtt{open}, \mathtt{closing}, \mathtt{count}).$$

Finally, let $X = [1, T] \setminus [1, \mathtt{start}]$ and $\mathtt{current} = ((v'_1)^{\prod_{i \in X} e_i}, \ldots, (v'_{\mathtt{len}})^{\prod_{i \in X} e_i})$.

Claim 2 *The Program correctness condition of Definition 3 holds for our construction.* (Proof of this claim is given in the full version.)

We briefly remark that all algorithms are polynomial time in the input. The concrete efficiency of the SeqProgram algorithm will not be as relevant to the performance of our forward secure signature schemes it will only be used in the proof of security and not in the actual construction.

6 An Efficient Scheme in the Random Oracle Model

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.) Our scheme will be parameterized by an RSA Sequencer as defined in Section 4 consisting of algorithms ($\mathsf{SeqSetup}$, $\mathsf{SeqUpdate}, \mathsf{SeqCurrent}$, $\mathsf{SeqShift}, \mathsf{SeqProgram}$).

Our initial scheme utilizes a random oracle G that we assume all algorithms have access to. For ease of exposition, we'll model the random oracle as a random function $G : \mathbb{Z}_N \times \{0,1\}^L \times [1,T] \rightarrow [0, 2^\lambda - 1]$ where N is an RSA modulus output from the global setup. We will often omit explicitly writing "mod N" and assume it implicitly when operations are performed on elements of $\mathbb{Z}_N^*$.

Hash Function to Prime Exponents. We make use of the hash function introduced in [13] and slightly refined in [14] to map integers to primes of an appropriate size. This hash function will not require the random oracle heuristic. The hash function $H : [1,T] \rightarrow \{0,1\}^{\lambda+1}$ takes as input a period $t \in [1,T]$ and output a prime between 2^λ and $2^{\lambda+1} - 1$. One samples the hash function by randomly choosing a K' for the PRF function $F : [1,T] \times [1, \lambda \cdot (\lambda^2 + \lambda)] \rightarrow \{0,1\}^\lambda$, a random $c \in \{0,1\}^\lambda$ as well as an arbitrary prime e_{default} between 2^λ and $2^{\lambda+1} - 1$. We let $K = (K', c, e_{\text{default}})$.

We describe how to compute $H_K(t)$. For $i = 1$ 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}.[8] This computation returns the smallest i such that $2^\lambda + y_i$ is a prime. Notationally, for $t \in [1,T]$ we will let $e_t = H_K(t)$.

We will use this hash function in this section and Section 7. For notational convenience, we will sometimes have algorithms pass a sampled key K instead of the description of the entire function H_K.

6.1 Construction

$\mathsf{Setup}(1^\lambda, 1^T)$ First, 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, the setup algorithm samples a hash function key K according to the description above. It follows by computing

$$\mathtt{state}_{\mathsf{pp}} = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=1}, g).^9$$

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 $\mathsf{pp} = (T, N, Y, K, \mathtt{state}_{\mathsf{pp}})$.

[8] The e_{default} value is included to guarantee that $H_K()$ returns some value for each input, but we have chosen the search space so that e_{default} is only returned with negligible probability.

[9] For convenience, we pass the key K to $\mathsf{SeqSetup}$ with the assumption that it implicitly describes H_K.

$\mathsf{KeyGen}(\mathsf{pp})$ The algorithm parses $\mathsf{pp} = (T, N, Y, K, \mathtt{state}_{\mathsf{pp}})$. It chooses a random integer u in $[1, N]$. It computes $\mathtt{state}_1 = \mathsf{SeqShift}(\mathtt{state}_{\mathsf{pp}}, u)$, $U = Y^u \bmod N$ and $e_1 = H_K(1)$. It sets $\mathsf{sk}_1 = (\mathtt{state}_1, e_1, 1)$ and $\mathsf{pk} = U$.

$\mathsf{Update}(\mathsf{pp}, \mathsf{sk}_t = (\mathtt{state}_t, e_t, t))$ The update algorithm computes $\mathtt{state}_{t+1} = \mathsf{SeqUpdate}(\mathtt{state}_t)$ and computes the prime $e_{t+1} = H_K(t+1)$ using pp. It outputs the new secret key as $\mathsf{sk}_{t+1} = (\mathtt{state}_{t+1}, e_{t+1}, t+1)$.

$\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}_t = (\mathtt{state}_t, e_t, t), M)$ The signing algorithm first computes $s = \mathsf{SeqCurrent}(\mathtt{state}_t)$.[10] It next chooses a random $r \in \mathbb{Z}_N^*$ and computes $\sigma_2 = G(r^{e_t} \bmod N, M, t)$. It then computes $\sigma_1 = r \cdot s^{\sigma_2}$. The signature for period t is output as $\sigma = (\sigma_1, \sigma_2)$.

$\mathsf{Verify}(\mathsf{pp}, \mathsf{pk} = U, M, t, \sigma = (\sigma_1, \sigma_2))$ The verification algorithm rejects if $\sigma_1 = 0 \bmod N$; otherwise it first computes the prime $e_t = H_K(t)$ using pp. It then computes $a = \sigma_1^{e_t}/(U^{\sigma_2})$ and outputs 1 to accept if and only if $G(a, M, t) \stackrel{?}{=} \sigma_2$.

Theorem 3. *If the RSA assumption (Assumption 1) holds, F is a secure pseudorandom function and G is modeled as a random oracle, then the Section 6.1 key-evolving signature construction is forward secure according to Definition 2.*

We prove this theorem in the full version via a series of 15 games. Correctness and efficiency analyses appear in the full version.

7 Streamlined Signatures in the Standard Model

We describe a scheme that is provably secure in the standard model with the restriction that the key must be updated after each signing (the scheme of the previous section does not share this restriction). This represents the best forward security practice assuming the underlying sign and update operations are efficient enough to support it. Our systems will be designed to provide practically efficient key generation, signing and update. Moreover we choose a signature structure that is optimized to provide as short a signature as possible. We achieve this by avoiding an RSA-based Chameleon hash as discussed in the introduction.

If more than one signature is issued during a time period t, the forward security property guarantees that all signatures issued before t remain secure. Moreover, for our particular construction, we claim that all signatures issued after t would remain secure as well. Informally, to see this, observe that each period t' is associated with a unique prime $e_{t'}$. Obtaining two signatures associated with the e_t-root could allow the adversary to produce additional signatures for time period t; however, it should not give the adversary any advantage in taking e_t'-roots for any $t' \neq t$. Indeed, we rely on this property to prove forward security. Thus, while single sign, our construction appears rather optimal in terms of mitigating the damage done if a user accidentally violates this restraint: she compromises signatures *only* for the time period for which she over-signed.

[10] Technically, $\mathsf{SeqCurrent}$ returns a tuple of length $\mathtt{len}$, since $\mathtt{len} = 1$ in this case, we allow $\mathsf{SeqCurrent}$ to return s instead of (s).

7.1 Construction

As before, 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.) Our scheme will be parameterized by an RSA Sequencer as defined in Section 4 consisting of algorithms ($\mathsf{SeqSetup}$, $\mathsf{SeqUpdate}$, $\mathsf{SeqCurrent}$, $\mathsf{SeqShift}$, $\mathsf{SeqProgram}$. In addition, it will use the same hashing function H to prime exponents as in Section 6.

Let $f : \mathbb{Z} \to \mathbb{Z}$ be a function such that $f(\lambda)/2^\lambda$ is negligible in λ. In this construction, 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 $k \cdot \ell = L$. Here, we will require that $2^\ell \leq f(\lambda)$. In our evaluation in Section 8, we will explore the performance impact of a various choices for the system parameters.

$\mathsf{Setup}(1^\lambda, 1^T)$ First, 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, the setup samples a hash function key K according of the description at the start of Section 6. It follows by computing

$$\mathtt{state}_{\mathsf{pp}} = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_1 = g, v_2 = g, \ldots, v_{\mathtt{len}} = g)).$$

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 $\mathsf{pp} = (T, N, Y, K, \mathtt{state}_{\mathsf{pp}})$.

$\mathsf{KeyGen}(\mathsf{pp})$ The algorithm retrieves Y from the pp. It chooses random integers $(u_0, u_1, \ldots, u_k, \tilde{u})$ in $[1, N]^{k+2}$. It computes $\mathtt{state}_1 = \mathsf{SeqShift}(\mathtt{state}_{\mathsf{pp}}, (u_0, u_1, \ldots, u_k, \tilde{u}))$. Next, for $i \in [0, k]$, it computes $U_i = Y^{u_i} \mod N$ and $\tilde{U} = Y^{\tilde{u}} \mod N$. It computes $e_1 = H_K(1)$. It sets $\mathsf{sk}_1 = (\mathtt{state}_1, e_1, 1)$ and $\mathsf{pk} = (U_0, U_1, \ldots, U_k, \tilde{U})$.

$\mathsf{Update}(\mathsf{pp}, \mathsf{sk}_t = (\mathtt{state}_t, e_t, t))$ The update algorithm computes $\mathtt{state}_{t+1} = \mathsf{SeqUpdate}(\mathtt{state}_t)$ and computes the prime $e_{t+1} = H_K(t+1)$ using pp. It outputs the new secret key as $\mathsf{sk}_{t+1} = (\mathtt{state}_{t+1}, e_{t+1}, t+1)$.

$\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}_t = (\mathtt{state}_t, e_t, t), M)$ The signing algorithm parses the $L = (\ell k)$-bit message $M = m_1 | m_2 | \ldots | m_k$, where each m_i contains ℓ-bits. Then it retrieves $(s_0, s_1, \ldots, s_k, \tilde{s}) = \mathsf{SeqCurrent}(\mathtt{state}_t)$. Next, it chooses random integer $r \in [0, 2^\lambda - f(\lambda)]$. The signature is generated as $\sigma = (\sigma_1, \sigma_2) = (s_0 \cdot \tilde{s}^r \cdot \prod_{j=1}^{k} s_j^{m_j},\ r)$.

$\mathsf{Verify}(\mathsf{pp}, \mathsf{pk}, M, t, \sigma = (\sigma_1, \sigma_2))$ Let $\mathsf{pk} = (U_0, \ldots, U_k, \tilde{U})$ and $M = m_1 | \ldots | m_k$. The verification first computes the prime $e_t = H_K(t)$ using pp. It accepts if and only if $0 \leq \sigma_2 \leq 2^\lambda - f(\lambda)$ and $\sigma_1^{e_t} \stackrel{?}{=} U_0 \cdot \tilde{U}^{\sigma_2} \cdot \prod_{j=1}^{k} U_j^{m_j}$.

Theorem 4. *If the RSA assumption (Assumption 1) holds and F is a secure pseudorandom function, then the Section 7.1 key-evolving signature construction is single-sign forward secure.*

Correctness, efficiency and proof of this theorem appear in the full version.

8 Performance Evaluation

We now analyze the performance of the two main forward-secure schemes presented: the random oracle based construction from Section 6 and the standard model construction from Section 7. The latter has the single sign restriction, however, our key update operations will be cheap enough to support a high rate of signing or one can use the hybrid certificate method discussed before.

For both constructions, we consider a 2048-bit RSA modulus N. To perform the timing evaluations in Figures 2 and 3, we utilized the high-performance NTL number theory library in C++ v10.5.0 by Victor Shoup [21]. Averaged over 10,000 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. 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 1.

Operation	$\mathbb{P}_{1024}$	$\mathbb{P}_{337}$	$\mathbb{P}_{113}$	$\mathbb{P}_{82}$	$\mathbb{P}_{81}$	$\mathbb{E}_{2048}$	$\mathbb{E}_{337}$	$\mathbb{E}_{336}$	$\mathbb{E}_{256}$
Time (ms)	28.533	1.759	0.365	0.317	0.302	4.700	0.815	0.808	0.638

Operation	$\mathbb{E}_{113}$	$\mathbb{E}_{112}$	$\mathbb{E}_{82}$	$\mathbb{E}_{81}$	$\mathbb{E}_{80}$	$\mathbb{E}_{32}$	$\mathbb{M}$
Time (ms)	0.305	0.299	0.226	0.217	0.211	0.098	0.001

Fig. 1. Time recorded in milliseconds for the above operations are averaged over 10,000 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.

For the Section 6 (Random Oracle) timing estimates in Figure 2, the message space is arbitrary, since the message is hashed as an input to the random oracle G. We set the maximum output length of G to be 80 bits. (Recall from our proof of security that an additive loss factor of 2^{-80} comes from the probability that the attacker receives the same challenge value from two forks of the security game at q^*.) Since the prime exponent must be larger than this output of G, we set it to be 81 bits.[11] These evaluations will be considered for a maximum number of periods of $T \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}, 2^{32}\}$.[12] The Setup algorithm computes the modular multiplications with respect to $\phi(N)$ while the other algorithms due

[11] The parameters given for this and the standard model scheme evaluation do not have a total correspondence to the scheme description, e.g., using 81-bit e values technically requires a variant of the RSA assumption with smaller exponents. We also do not attempt to set the modulus size to match the security loss of our reductions. It is unknown if this loss can be utilized by an attacker and we leave it as future work to deduce an optimally tight reduction. Our focus here is to give the reader a sense of the relative performance of the schemes for reasonable parameters.

[12] Technically, $T = 2^{\mathtt{levels}+1} - 2$ (see Section 5), we ignore the small constants.

Sec. 6 Alg.	Operation Count	Time when $T =$ 2^{12}	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}
Setup	$T \cdot \mathbb{P}_{\lvert e\rvert} + 2\lg T \cdot \mathbb{E}_{\lvert N\rvert} + (2T \lg T) \cdot \mathbb{M}$	1.45s	22.03s	5.98m	1.63h	1.11d	18.16d
KeyGen	$1 \cdot \mathbb{P}_{\lvert e\rvert} + (2\lg T + 1) \cdot \mathbb{E}_{\lvert N\rvert}$	0.12s	0.16s	0.19s	0.23s	0.27s	0.31s
Update	$\lg T \cdot \mathbb{P}_{\lvert e\rvert} + \lg T \cdot \mathbb{E}_{\lvert e\rvert}$	6.24ms	8.32ms	10.40ms	12.48ms	14.56ms	16.64ms
Sign	$1 \cdot \mathbb{E}_{\lvert e\rvert} + 1 \cdot \mathbb{E}_{\lvert \sigma_2\rvert} + 1 \cdot \mathbb{M}$	0.43ms	0.43ms	0.43ms	0.43ms	0.43ms	0.43ms
Verify	$1 \cdot (\mathbb{P}_{\lvert e\rvert} + \mathbb{E}_{\lvert e\rvert} + \mathbb{E}_{\lvert \sigma_2\rvert} + \mathbb{M})$	0.73ms	0.73ms	0.73ms	0.73ms	0.73ms	0.73ms

Fig. 2. Running Time Estimate for the Section 6 (Random Oracle) Scheme with a 2048-bit N. Let $\mathbb{P}_{|e|}$ 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. T is the maximum number of time periods supported by the forward-secure scheme. We set $|e| = 81$ bits to be the size of the prime exponents and $|\sigma_2| = 80$ bits to be the maximum size of the output of G. We set the message space length L to be an arbitrary polynomial function of λ. Times are calculated by taking the average time for an operation (see Figure 1) and summing up the total times of each operation. Let ms denote milliseconds, s denote seconds, m denote minutes, h denote hours, and d denote days.

so with respect to N. However, since $\phi(N)$ is very close to N, we treat both of these the same (i.e., at 2048 bits); we do this in the timing of both schemes. In Sign and Verify, we do not consider the time to compute the random oracle G.

For the Section 7 (Standard Model) timing estimates in Figure 3, the messages space is $L = k \cdot \ell = 256$, where messages are broken into k chunks each of ℓ bits. We consider three different settings of k and ℓ, keeping the prime exponent associated with that setting to be at least one bit larger than the size of the message chunks. Here we do not recommend allowing the size of the prime exponents to fall below 80 bits to avoid collisions.

8.1 Some Comparisons and Conclusions

We make a few brief remarks and observations. First, if one wants to support a high number of key updates, then it is desirable to offload much of the cost of the key generation algorithm to a one time global setup. Having a one time global setup that takes a few days might be reasonable [13], while incurring such a cost on a per user key setup basis could be prohibitive. With one exception ($k = 256$ in Figure 3) all individual key generation times are at most a few seconds. One question is how much trust needs to be placed into one party for a global setup. Fortunately, for our constructions, the answer is favorable. First, there are efficient algorithms for generating RSA moduli that distribute trust across multiple parties [8], so the shared N could be computed this way. Second, once the RSA modulus plus generator g and RSA exponent hashing key are chosen, the rest of the RSA sequencer computation can be done deterministically

[13] This could be further reduced by using a faster computer and/or parallelizing.

Sec.7 Alg.	Operation Count	Parameters k	$\|e\|$	$\|\sigma_2\|$	Time when $T =$ 2^{12}	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}
Setup	$T \cdot \mathbb{P}_{\|e\|}+$	1	337	336	7.41s	1.96m	31.42m	8.42h	5.63d	90.54d
	$2\lg T \cdot \mathbb{E}_{\|N\|}+$	8	113	112	1.70s	26.11s	7.06m	1.92h	1.30d	21.25d
	$(2T \lg T) \cdot \mathbb{M}$	256	82	81	1.51s	23.0s	6.23m	1.70h	1.16d	18.89d
KeyGen	$\mathbb{P}_{\|e\|}+(k+2)\cdot$	1	337	336	0.35s	0.47s	0.58s	0.69s	0.81s	0.92s
	$(2\lg T + 1)\cdot$	8	113	112	1.17s	1.55s	1.93s	2.30s	2.68s	3.06s
	$\mathbb{E}_{\|N\|}$	256	82	81	30.32s	40.02s	49.72s	59.42s	1.15m	1.31m
Update	$\lg T \cdot \mathbb{P}_{\|e\|}+$	1	337	336	50.46ms	67.28ms	84.10ms	0.10s	0.12s	0.13s
	$(k+2)\lg T\cdot$	8	113	112	41.01ms	54.67ms	68.34ms	82.01ms	95.68ms	0.11s
	$\mathbb{E}_{\|e\|}$	256	82	81	0.70s	0.94s	1.17s	1.41s	1.64s	1.87s
Sign	$k \cdot \mathbb{E}_\ell + \mathbb{E}_{\|\sigma_2\|}$	1	337	336	1.45ms	1.45ms	1.45ms	1.45ms	1.45ms	1.45ms
	$+(k+1) \cdot \mathbb{M}$	8	113	112	1.09ms	1.09ms	1.09ms	1.09ms	1.09ms	1.09ms
		256	82	81	0.47ms	0.47ms	0.47ms	0.47ms	0.47ms	0.47ms
Verify	$\mathbb{P}_{\|e\|} + k \cdot \mathbb{E}_\ell +$	1	337	336	4.02ms	4.02ms	4.02ms	4.02ms	4.02ms	4.02ms
	$\mathbb{E}_{\|\sigma_2\|} + \mathbb{E}_{\|e\|}+$	8	113	112	1.76ms	1.76ms	1.76ms	1.76ms	1.76ms	1.76ms
	$(k+1) \cdot \mathbb{M}$	256	82	81	1.01ms	1.01ms	1.01ms	1.01ms	1.01ms	1.01ms

Fig. 3. Running Time Estimate for the Section 7 Scheme with a 2048-bit N. Let $\mathbb{P}_{|e|}$ 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. T is the maximum number of time periods supported by the forward-secure scheme. We set the message space length $L = k \cdot \ell = 256$ bits. Times are calculated by taking the average time for an operation (see Figure 1) and summing up the total times of each operation. Let ms denote milliseconds, s denote seconds, m denote minutes, h denote hours, and d denote days.

and without knowledge of any secrets. Thus, a few additional parties could audit the rest of the global setup assuming they were willing to absorb the cost.

We now move to discussing the viability of our standard model construction. We focus on the setting of $k = 8$ as a representative that seems to provide the best tradeoffs of the three settings explored. Here the global setup time will take around 7 minutes if we want to support up to a million key updates and will take on the order of a few days if we want to push this to around a billion updates. The global setup cost here is close to that of the random oracle counterpart. Individual key generation takes between 1 and 3 seconds depending of the number of time periods supported. The time cost of signing and verifying does not scale with T, the max number of time periods, and these incur respective costs of 1.09ms and 1.76ms. Signatures are 0.26KB regardless of T.

The important measurement to zoom in on is key update. This algorithm however, is more expensive and ranges in cost from 50ms to around 110ms depending on T. Since (in the basic mode) one is allowed a single signature per key update, it will serve as the bottleneck for how many signatures one can produce. In this case the number is between 10 to 20 per second. In many applications this is likely sufficient. However, if one needs to generate signatures at a faster rate, then she will need to move to the certificate approach where the tradeoff will be

Sec. 6 Item	Element Count	Space when $T =$ 2^{12}	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}
pp	$((2\lg T)+1)\mathbb{Z}_N$	6.25K	8.25K	10.25K	12.25K	14.25K	16.25K
pk	$1\mathbb{Z}_N$	0.25K	0.25K	0.25K	0.25K	0.25K	0.25K
sk	$(2\lg T)\mathbb{Z}_N + 1\|e\|$	6.0K	8.0K	10.0K	12.0K	14.0K	16.0K
σ	$1\mathbb{Z}_N + 1\|\sigma_2\|$	0.26K	0.26K	0.26K	0.26K	0.26K	0.26K

Fig. 4. Space Evaluation for Section 6 (Random Oracle) Scheme. Let the modulus be a 2048-bit N. Let K denote a kilobyte (2^{10} bytes). T is the maximum number of time periods supported by the forward-secure scheme. We consider $|e| = 81$ bits to be the size of the exponents and $|\sigma_2| = 80$ bits to be the maximum size of the output of G. The public parameters and keys omit the descriptions of T, N and the hash function H_K. For the public parameters, all $\mathtt{len} = k + 2$ generators are the same, so we use an optimization detailed in the full version.

that the signature size increases to accommodate the additional signature (e.g., certificate) plus temporary public key description.

Finally, we observe that for most of our standard model algorithms parallelization can be used for speedup in fairly obvious ways. In particular in key update and key generation there are $\lg(T)$ levels as well as $k+2$ message segments and one can partition the computation along these lines.

References

1. Michel Abdalla, Fabrice Benhamouda, and David Pointcheval. On the tightness of forward-secure signature reductions. *J. Cryptology*, 32(1):84–150, 2019.
2. Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In *Advances in Cryptology - ASIACRYPT*, pages 116–129, 2000.
3. Ross Anderson. Invited Lecture. Fourth Annual Conference on Computer and Communications Security, ACM, 1997.
4. Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *Advances in Cryptology - CRYPTO*, pages 431–448, 1999.
5. Mihir Bellare and Todor Ristov. Hash functions from sigma protocols and improvements to VSH. In *ASIACRYPT*, pages 125–142, 2008.
6. Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT*, pages 274–285, 1993.
7. Dan Boneh and Xavier Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In *EUROCRYPT*, volume 3027, pages 223–238, 2004.
8. Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. *J. ACM*, 48(4):702–722, 2001.
9. Jan Camenisch and Maciej Koprowski. Fine-grained forward-secure signature schemes without random oracles. *Discrete Applied Mathematics*, 154(2):175–188, 2006.
10. Dennis Fisher. Final Report on DigiNotar Hack Shows Total Compromise of CA Servers. Threatpost, 10/31/12. At `https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/`.

Sec.7 Item	Element Count	Parameters k	$\|e\|$	$\|\sigma_2\|$	Space when $T =$ 2^{12}	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}
pp	$((2\lg T)+1)\mathbb{Z}_N$	any	any	any	6.25K	8.25K	10.25K	12.25K	14.25K	16.25K
pk	$(k+2)\mathbb{Z}_N$	1	337	336	0.75K	0.75K	0.75K	0.75K	0.75K	0.75K
		8	113	112	2.5K	2.5K	2.5K	2.5K	2.5K	2.5K
		256	82	81	64.5K	64.5K	64.5K	64.5K	64.5K	64.5K
sk	$(k+2)(2\lg T)\mathbb{Z}_N$ $+1\|e\|$	1	337	336	18.0K	24.0K	30.0K	36.0K	42.0K	48.0K
		8	113	112	60.0K	80.0K	100.0K	120.0K	140.0K	160.0K
		256	82	81	1.51M	2.01M	2.52M	3.02M	3.53M	4.03M
σ	$1\mathbb{Z}_N + 1\|\sigma_2\|$	1	337	336	0.29K	0.29K	0.29K	0.29K	0.29K	0.29K
		8	113	112	0.26K	0.26K	0.26K	0.26K	0.26K	0.26K
		256	82	81	0.26K	0.26K	0.26K	0.26K	0.26K	0.26K

Fig. 5. Space Evaluation for Section 7 Scheme. Let the modulus be a 2048-bit N. Let K denote a kilobyte (2^{10} bytes) and M denote a megabyte (2^{20} bytes). T is the maximum number of time periods supported by the forward-secure scheme. The public parameters and keys omit the descriptions of T, N and the hash function H_K. For the public parameters, all $\texttt{len} = k+2$ generators are the same, so we use an optimization detailed in the full version.

11. Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, 1988.
12. Stefanie Hoffman. RSA SecureID Breach Costs EMC $66 Million. CRN Magazine, July 28, 2011. At `http://www.crn.com/news/security/231002862/rsa-secureid-breach-costs-emc-66-million.htm`.
13. Susan Hohenberger and Brent Waters. Short and stateless signatures from the RSA assumption. In *CRYPTO '09*, volume 5677 of LNCS, pages 654–670, 2009.
14. Susan Hohenberger and Brent Waters. Synchronized aggregate signatures from the RSA assumption. In *EUROCRYPT*, volume 10821, pages 197–229, 2018.
15. Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In *Advances in Cryptology - CRYPTO*, pages 332–354, 2001.
16. Anton Kozlov and Leonid Reyzin. Forward-secure signatures with fast key update. In *Security in Communication Networks*, pages 241–256, 2002.
17. Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *ACM Conference on Computer and Comm. Security*, pages 108–115, 2000.
18. Tal Malkin, Daniele Micciancio, and Sara K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Advances in Cryptology - EUROCRYPT*, pages 400–417, 2002.
19. Payman Mohassel. One-time signatures and chameleon hash functions. In *Selected Areas in Cryptography - 17th International Workshop, SAC*, pages 302–319, 2010.
20. 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.
21. Victor Shoup. NTL: A Library for doing Number Theory, v10.5.0, 2017. Available at `http://www.shoup.net/ntl/`.
22. Dawn Xiaodong Song. Practical forward secure group signature schemes. In *ACM Conference on Computer and Communications Security*, pages 225–234, 2001.