New Methods and Abstractions for RSA-Based Forward Secure Signatures

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

Brent Waters[†]
University of Texas at Austin and NTT Research
bwaters@cs.utexas.edu

July 10, 2020

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. 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. Our sequencer abstraction also has certain re-randomization properties that allow for constructing forward-secure signatures with a single trusted setup that takes T time and 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 is much faster (after the one-time setup). We then move on to designing a standard model scheme. This abstraction and illustration of how to use it may 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 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.

[*]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.

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 [22]. Also in 2011, the certificate authority DigiNotar was compromised and found that several rogue certificates for companies such as Google were issued in Iran [19]. 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 [19].

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 [6, 8]. 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 $t + 1$.[1] 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 [6] and Bellare and Miner [8], 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.

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 $h \pmod N$. This technique was used in many other works including Camenisch and Koprowski [14] 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., [42, 14, 3]) 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

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

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.

RSA Sequencers. The conceptual starting point of our work is a methodology for separating out the tasks of storing and updating RSA-based private key material from the design of the core signature scheme, which we believe will be useful in other settings as well. We capture this separation formally with the introduction of an abstraction we call an RSA sequencer. Informally, the job of an RSA sequencer is to store roots of a public key U, such 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 first focus on building a sequencer that efficiently stores such values, in a forward secure manner and with a desire to achieve better setup times than other comparable solutions. Next, we can focus on the design and mechanics of different signature schemes without worrying about how such a sequencer is implemented. A more detailed technical overview follows.

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 $V_1^{1/e_t},\ldots,V_{\mathtt{len}}^{1/e_t}$.[2]

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 $\mathtt{state}_t$ 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 $\mathsf{SeqShift}$ algorithm. Looking forward in our RSA Sequencer construction the $\mathsf{SeqShift}$ will be a significantly cheaper function to call as its computation time will scale proportionally to $\lg(T)$, while the $\mathsf{SeqSetup}$ algorithm will run in time proportional

[2]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.

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 $\texttt{start} \in [1, T]$, consider a tuple $v'_1 = v_1^{\prod_{i \in [1, \texttt{start}-1]} e_i}, \ldots, v'_{\texttt{len}} = v_{\texttt{len}}^{\prod_{i \in [1, \texttt{start}-1]} e_i}$. Then $\mathsf{SeqProgram}(N, 1^T, H, 1^{\texttt{len}}, (v'_1, \ldots, v'_{\texttt{len}}), \texttt{start})$ produces the same output as $\mathsf{SeqSetup}(N, 1^T, H, 1^{\texttt{len}}, (v_1, \ldots, v_{\texttt{len}}))$ followed by $\texttt{start} - 1$ iterative calls to SeqUpdate. Intuitively, the semantics of SeqProgram provide an interface to generate the $\texttt{start}$-th private key without knowing any of the first $\texttt{start} - 1$ roots of $V_1, \ldots, V_{\texttt{len}}$.

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 [27] 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 [28] for forward secure signatures. The RSA Sequencer bears some history and resemblance to accumulators [10], but has different goals, algorithms and constructions.

In our construction the (optimized version of the) SeqSetup algorithm makes T calls to H and performs $T \cdot \texttt{len}$ 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 $2 \cdot \texttt{len}$ 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 We now turn to the task of designing forward secure signature systems using the RSA Sequencer interface. We actually begin with providing a random oracle construction. We do this for two reasons. First, we use this first construction to establish a baseline of comparison for our standard model schemes. Second, this first construction gives a basic example of how to apply the RSA sequencer abstraction that will be instructive for the later scheme. The system we present will have an underlying signature technique similar to the Guillou-Quisquater (GQ) [21] signature scheme. When instantiated with our logarithmic-update sequencer construction of Section 5, it will result in a forward secure signature scheme with underlying mechanics and performance close to the efficient Itkis-Reyzin [28] scheme under the pebbling optimization. However, one important difference is our use of a SeqShift algorithm that allows us to generate user key pairs quickly from a given set of global public parameters, while public key creation is expensive in their scheme. For a large number of time periods such as $T = 2^{32}$ this can mean the difference between key generation taking tens of milliseconds versus over twenty days!

In our Section 6 signature scheme, the (global) Setup algorithm will take as input a security parameter and number of time periods T. Next, it will choose an RSA modulus N with a generator g of QR_N and a key K to a hash function H where H_K maps periods $t \in [1, T]$ to RSA exponents. (See Section 9.1 for a discussion of how to reduce the trust needed in the party (or parties) running Setup.) We use a construction of H from [26] that allows us to program in as output a given RSA exponent e^* for a chosen input t^*. Next, let $E = \prod_{j=1}^{T} e_j \mod \phi(N)$ and $Y = g^E \mod N$. Finally, the setup algorithm computes $\mathtt{state}_{\mathsf{pp}} = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=1}, g)$. The public parameters are $\mathsf{pp} = (T, N, Y, K, \mathtt{state}_{\mathsf{pp}})$. Using an optimization (see Section 5), the dominant cost of this operation is T calls to H which amount to T prime searches for security parameter length primes.

The $\mathsf{KeyGen}(\mathsf{pp})$ algorithm is relatively straightforward and inexpensive. The algorithm simply chooses an exponent $u \in [1, N]$ and sets $U = Y^u$ as the public key and computes the secret key as $\mathsf{SeqShift}(\mathtt{state}_{\mathsf{pp}}, u)$. Since the original sequencer $\mathtt{state}_{\mathsf{pp}}$ allowed for computing (arbitrary $e_j \neq e_t$) roots of the value Y^{1/e_t} at period t, the new state will allow for computing roots of $Y^u = U$. If $\mathtt{state}_t$ is the secret key state at period t, one computes the next state as $\mathtt{state}_{t+1} = \mathsf{SeqUpdate}(\mathtt{state}_t)$.

If a signature scheme is at period t with $\mathtt{state}_t$, to sign a message simply call $\mathsf{SeqCurrent}(\mathtt{state}_t)$ to get $s = U^{1/e_t}$. The signature is then computed using the GQ scheme [21] with a random oracle G. The algorithm chooses a random $r \in \mathbb{Z}_N^*$, computes $\sigma_2 = G(r^{e_t} \mod N, M, t)$ and then computes $\sigma_1 = r \cdot s^{\sigma_2}$. The signature is (σ_1, σ_2). The verification algorithm simply checks that the signature matches the correct form.

The proof of security leverages the RSA Sequencer model to abstract away many details. Our proof is organized around a sequence of games. At a high level, it first uses $\mathsf{SeqShift}$ correctness to switch from using the shift operation to directly creating the sequencer state from a group element. Next, it changes from giving the corrupted key by updates to using the $\mathsf{SeqProgram}$ algorithm to directly create it at some time period b (corresponding to the "break in" by the forward secure adversary) which is greater than the forgery period t^*. This alleviates the reduction from needing to compute the e_{t^*}-th root of the public key U. We defer other details of the proof to the main body. These include embedding an RSA challenge and an application of a "forking" lemma that are similar to prior works.

Streamlined Signatures in the Standard Model. We now show how we can move to the standard model by changing the design of the signature scheme, but keeping the key storage mechanism very close. In this construction, we will be 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.

We show how to develop a signature structure that is provably secure in the standard model. Consider the hash function $U_0 \cdot \prod_{j=1}^{k} U_j^{m_j}$ for public parameters $U_0, U_1, \ldots, U_k$ and input $M = m_1|m_2|\ldots|m_k$. A signature on message $M = m_1|m_2|\ldots|m_k$ for time period t is simply the e_t-th root of this group element. While signature structures of similar form have been seen in the literature, they have only be proven weakly secure — that is against a model where the attacker must a priori declare which messages it will sign query. The reason for weak security is that the reduction algorithm will guess a cancellation of the RSA challenge h such that if (and only if) the

j^*-th message segment is m_{j^*} all "h terms" will cancel out in the hash evaluation.

The remedy for moving to full security, of course, is to instead use the scheme to sign a chameleon hash [31] of the message so that the reduction to the RSA assumption will know a priori what message is being signed. However, an RSA-based chameleon hash [7, 9] will encumber another element of $\mathbb{Z}_N^*$ and double the signature size. In our scheme we choose an additional random value r and multiply in $\tilde{U}^r$ into the hash function computation. The goal is to dynamically fit r to cancel the h terms in the hash evaluation for the queried message, but so that it is hard for the attacker to find randomness r' that will achieve cancellation for a different message segment value. The analysis is complicated by the fact that the group order is unknown in an RSA reduction, which precludes a basic pairwise independence analysis.

Fischlin [18] has a signature scheme that incorporates randomness in a similar manner. This scheme is further analyzed by Hofheinz and Kiltz [24] through the lens of what they call randomized programmable hash functions.

We describe our scheme and analysis in more detail below. We consider messages of length L and break them into k "chunks", each of length ℓ. The global setup algorithm will choose the RSA modulus N and hashing key K as before. This time it will run $\mathtt{state}_{\mathsf{pp}} = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_1 = g, v_2 = g, \ldots, v_{\mathtt{len}} = g))$. Nominally, this will take a factor of $(k+2)$ longer than the $\mathsf{SeqSetup}$ for $\mathtt{len} = 1$ relative to the random oracle scheme; however since v_i is set to be the same generator g for all i, we can avoid the redundant work and the $k+2$ factor if we break abstraction slightly. It also computes $E = \prod_{j=1}^{T} e_j \mod \phi(N)$ and $Y = g^E \mod N$.

To generate a key pair, the KeyGen algorithm first chooses integers $(u_0, u_1, \ldots, u_k, \tilde{u})$ in $[1, N]^{k+2}$. Then 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 $\tilde{U} = Y^{\tilde{u}} \mod N$. Roughly, this is close to a factor of $(k+2)$ times the computational and key storage cost of the random oracle scheme. For the case of $k = 1$, the difference is a factor of 3. However, the RSA exponents need to be significantly bigger than the chunk sizes so there can be reasons to push toward slightly larger k and smaller chunk sizes. (We'll explore some tradeoffs in our performance analysis in Section 9.)

The signing algorithm first parses the $L = (\ell k)$-bit message M as $M = m_1|m_2|\ldots|m_k$. It retrieves $(s_0, s_1, \ldots, s_k, \tilde{s}) = \mathsf{SeqCurrent}(\mathtt{state}_t)$. Next, it chooses random integer r. The signature is generated as $\sigma = (\sigma_1, \sigma_2) = (s_0 \cdot \tilde{s}^r \cdot \prod_{j=1}^{k} s_j^{m_j},\ r)$. The value r must be chosen from a range such that the largest possible chunk size is statistically insignificant compared to r. For example, if $k = 1$ and we had messages of $L = 256$ bits (say from the output of SHA-256), then we might choose r to be a random $256 + 80 = 336$ bit integer. Finally, the verification equation tests the well-formedness of the signatures using the U_i values from the public key.

The proof of security follows the random oracle counterpart closely in terms of how private key corruption is simulated using the $\mathsf{SeqProgram}$ algorithm as well as how signatures are generated on time periods other than t^*. The proof departs in how signatures are generated for the time period t^*. Our main reduction will first guess an index $\alpha \in [1, k]$ where the forgery message differs from the queried message in the message segment α at time period t^*. (If the guess is wrong the reduction aborts.) It also choose a random value $\beta \in \{0, 1\}$. Intuitively, for an RSA challenge h we will choose a random value $d \in [0, 2^\lambda]$ and set the parameters as $U_0 = (h^{E'})^{-d}$, $U_\alpha = (h^{E'})^{-\beta}$ and $\tilde{U} = h^{E'}$ where $E' = E/e_{t^*}$. (We omit some additional terms that ensure that the public parameters in the reduction have the same distribution as in the real scheme.) When the attacker queries on a message at time period t^* with the segment m_α, we set $r = d + \beta m_\alpha$ which will ensure that all of the h terms cancel out and we can sign the message. Observe that since r is chosen from a much bigger range than the message chunk size, choosing from this distribution is statistically indistinguishable from choosing r from the original range. Now suppose we receive a forgery for $m_\alpha^* \neq m_\alpha$. The question is whether the σ_2^* we get from the attacker is such that $\sigma_2^* \neq d + \beta m_\alpha^*$; only in this case will we be able to extract an RSA solution. We argue that this will be the case at least 1/2 the time since the random bit β is statistically hidden from the attacker.

Removing the Single Sign Restriction and Other Variants. In Section 8, we point out a generic method to remove the single sign per update restriction. The transformation follows a path common in literature where the base forward-secure scheme signs as its message another verification key. 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. In this manner, the forward-secure signature algorithm is invoked once per update, but the regular scheme can generate many signatures. A tradeoff is that the signature size increases relative to the single sign approach.

The separation between our RSA sequencer abstraction for key storage and the core signature design leaves the door open for building other constructions. For example, one might apply the more general signature structures from Hofheniz, Jager and Kiltz [23] to explore different tradeoffs.

Performance Analysis. In Section 9, 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 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 even for our standard model construction. 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.

1.1 Related Work Discussion

The concept of forward-secure signatures was first put forward in a talk by Anderson [6] in 1997 to address a serious security weakness he saw in regular digital signatures: namely, that once a key is compromised, *every* signature generated by that key must be treated as invalid. The solution, forward-secure signatures, was given a formal treatment in 1999 by Bellare and Miner [8]. They also provided tree and Fiat-Shamir [17] based constructions satisfying this new definition. The tree-based construction builds a binary tree of certificates from any signature scheme where the leaves correspond to time periods. When T is the maximum number of time periods allowed by the scheme, this construction takes roughly a multiplicative factor of $O(\lg T)$ over the underlying signature scheme in terms of private key size, and signing and verification time. In their "hash-and-sign" construction in the random oracle model, they achieve short signatures with fast key update, but the verification is linear in T and the public and secret keys require many elements in $\mathbb{Z}_N$. Many new constructions followed. Malkin-Micciancio-Miner [34] explored new constructions and trade-offs for tree-based constructions, including achieving signing and verification times that require a logarithmic number of hash function evaluations (rather than signing and verification operations) at the cost of longer setup and key generation times. Some of these tree-based schemes are in the standard model. In Section 9.2, we compare our results to [34] and argue that our solution provides significantly shorter signatures and much faster key generation times.

Krawczyk [30] 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 [5] showed how to shorten the private keys in the "hash-and-sign" Bellare-Miner [8] construction in the random oracle model. Itkis and Reyzin [28] presented GQ-based signatures with "optimal" signing and verification in the random oracle model using a very elegant pebbling approach. Camenisch and Koprowski [14] 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 [29] 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 [9] with a transformation due to Mohassel [36]. 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.

Other interesting related works include a performance study of forward-secure signatures [16], an efficient bilinear maps based construction in the standard model [13], and tighter reductions [1, 2]. There have been many interesting extensions of forward security to threshold signatures [4], group signatures [42, 33], and signatures with untrusted update [13, 32].

2 Definitions

Following prior works [8, 28], 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 [8] 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 2.1 (Key-Evolving Signatures [8, 28]). *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 [8] and in turn built on the Goldwasser, Micali and Rivest [20] 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.[3] 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.[4] 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. $\mathsf{Verify}(\mathsf{pp}, \mathsf{pk}, m, t, \sigma) = 1$.

[3]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.

[4]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.

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.

Definition 2.2 (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 [11]) 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 [39] 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 $p'q'$. 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 $h = x^e \mod N$.*

A randomly chosen element in $\mathbb{Z}_N^*$ would be a quadratic residue 1/4-th of the time, so the restriction to focus on QR_N is for convenience and could be relaxed.

In our schemes, we will refer to and require a primality test, such as the efficient Miller-Rabin test [35, 38]. We will also make use of the following lemmas:

Lemma 3.2 (Shamir's Trick [40, 15])**.** *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 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 4.1 (RSA Sequencer)**.** *An RSA Sequencer consists of a tuple of* deterministic *algorithms* $(\mathsf{SeqSetup}, \mathsf{SeqUpdate}, \mathsf{SeqCurrent}, \mathsf{SeqShift}, \mathsf{SeqProgram})$ *such that:*

$\mathsf{SeqSetup}(N \in \mathbb{Z}, 1^T, H : \{1, \dots, T\} \to \mathbb{Z}, 1^{\texttt{len}}, (v_1, \dots, v_{\texttt{len}}) \in \mathbb{Z}_N^{\texttt{len}})$: 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 $\mathsf{SeqSetup}$ algorithm outputs a state value `state`.

$\mathsf{SeqUpdate}(\texttt{state})$: On input of a state value `state`, the $\mathsf{SeqUpdate}$ algorithm produces another value $\texttt{state}'$.

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

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

$\mathsf{SeqProgram}$ $(N \in \mathbb{Z}, 1^T, H : \{1 \dots, T\} \to \mathbb{Z}, 1^{\texttt{len}}, (v'_1, \dots, v'_{\texttt{len}}) \in \mathbb{Z}_N^{\texttt{len}}, \texttt{start} \in \{1, \dots, T\})$: 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 $\texttt{start} \in [1, T]$, the $\mathsf{SeqProgram}$ algorithm outputs a state value `state`.

We note that the $\mathsf{SeqProgram}$ 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 $\mathsf{SeqSetup}$ and $\mathsf{SeqProgram}$ 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 $\mathsf{SeqSetup}$) 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 $N \in \mathbb{Z}, T \in \mathbb{Z}, H : \{1 \dots, T\} \to \mathbb{Z}, \texttt{len} \in \mathbb{Z}, (v_1, \dots, v_{\texttt{len}}) \in \mathbb{Z}_N^{\texttt{len}}$, the following must hold: Let $\texttt{state}_1 = \mathsf{SeqSetup}(N, 1^T, H, 1^{\texttt{len}}, (v_1, \dots, v_{\texttt{len}}))$. For $t = 2$ to T, let $\texttt{state}_t =$
$\mathsf{SeqUpdate}(\texttt{state}_{t-1})$. Then for all $t \in [1, T]$, it must be that

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

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 $\mathsf{SeqShift}$ is operated immediately on the initial state output of $\mathsf{SeqSetup}$.

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 [27] and is also conceptually similar to the pebbling optimization from Itkis and Reyzin [28].

Let's recall the purpose of an RSA sequencer from the introduction. Again, 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 $\mathsf{SeqUpdate}$, the $\mathsf{SeqCurrent}$ 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 $\mathsf{SeqUpdate}$ the data structure can simply store $v^{\prod_{i \in [1,k]} e_i}$. In this manner the $\mathsf{SeqUpdate}$ algorithm only needs a single exponentiation to update the data structure, but the $\mathsf{SeqCurrent}$ 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 $\mathsf{SeqUpdate}$, the data structure will already have $V^{1/e_{k+1}}$ ready for retrieval. Moreover, the next $\mathsf{SeqUpdate}$ call will do a logarithmic amount of work that has the next one ready as well. Intuitively, each call to $\mathsf{SeqUpdate}$ will perform work that both applies to computing "nearby" roots as well as progress towards further out time periods. The description below gives these in detail and for the slightly more general case of 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 $\mathsf{SeqProgram}$ algorithm which we will circle back to at the end of the section.

$\mathsf{SeqSetup}(N, 1^T, H, 1^{\texttt{len}}, (v_1, \ldots, v_{\texttt{len}}))$ Initialize, sets $S_1, \ldots, S_{\texttt{levels}}$ to be empty. Then for $i = 2$ to $\texttt{levels}$ 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 the tuple $\texttt{current}$ as $\texttt{current} = (w_1^{e_2}, \ldots, w_{\texttt{len}}^{e_2})$.

The output is

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

$\mathsf{SeqUpdate}(\texttt{state})$ For $i = 1$ to $\texttt{levels}$, 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 $\texttt{open}$ value.[5]
- 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 $i = \texttt{levels}$ 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 $\texttt{state}' = \big(\texttt{index}', \texttt{current}', (S_1, \ldots, S_{\texttt{levels}})\big)$.

[5] 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 our analysis will show.

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

$\mathsf{SeqShift}(\mathtt{state}, (z_1, \ldots, z_{\mathtt{len}}))$ 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 $\mathtt{state}' = \big(\mathtt{index}, \mathtt{current}', (S_1, \ldots, S_{\mathtt{levels}})\big)$.

5.1 Efficiency

We remark on the efficiency of the above algorithms. Recall that $T = 2^{\mathtt{levels}+1} - 2$. If we ignore the minus 2 component, this means that $\mathtt{levels} = \lg(T) - 1$. A call to the $\mathsf{SeqUpdate}$ algorithm will invoke at most $\mathtt{levels}$ calls to H in the first phase of the algorithm. In addition, the first pass will invoke at most $\mathtt{levels} \cdot \mathtt{len}$ exponentiations in $\mathbb{Z}_N$. The size of the exponents will depend on the output length of H, which may vary with different choices of H. The second pass of the algorithm only involves shuffling around data and its computational costs will be dominated by the first. Likewise, the computational component of $\mathsf{SeqCurrent}$ is tiny since it simply returns a stored value.

In the $\mathsf{SeqSetup}$ algorithm we bound the computational costs. For each level there will be at most T calls to H and $T \cdot \mathtt{len}$ exponentiations in $\mathbb{Z}_N$ where again the size of the exponents is determined by the output length of H. Thus, there are at most $\mathtt{levels} \cdot T$ calls to H and $\mathtt{levels} \cdot T \cdot \mathtt{len}$ exponentiations.

Optimizations We mention two alternative optimizations. In the prior description the algorithm moved in sequential fashion by forming each S_i in turn. Instead one could loose the $\mathtt{levels} = \lg(T) - 1$ factor from the number of calls to H by forming these sets in parallel.

First let R_i be the range used to create the i-th set and define $X_i = [1, T] \setminus R_i$. For each set, the algorithm would initialize it with the group element tuple $(v_1, \ldots, v_{\mathtt{len}})$ (along with the appropriate $\mathtt{open}, \mathtt{closing}$ and $\mathtt{count}$ values). Then for $n = 1$ to T, compute $e_n = H(n)$. For each n, raise the group elements in S_i to e_n if and only if $n \in X_i$. In this way, each e_n value is computed only once.

The second optimization requires us to break our abstraction slightly. Suppose that the party running $\mathsf{SeqSetup}$ has chosen N to be an RSA modulus and knows $\phi(N)$. This will actually be the case for our forward-secure signature schemes of Sections 6 and 7. In this case for each $i \in \mathtt{levels}$, the algorithm first computes $\prod_{j \in X_i} e_j \mod \phi(N)$ and only exponentiates at the end. This will reduce the cost to be bounded by $T \cdot \mathtt{levels} \cdot \mathtt{len}$ multiplications in $\phi(N)$ and $2 \cdot \mathtt{levels} \cdot \mathtt{len}$ exponentiations in $\mathbb{Z}_N$.

5.2 Correctness Analysis

For our analysis, we establish a core lemma that describes the sequencer's state after a given number of update steps. With this is in place, we can prove the correctness conditions of our scheme.

Lemma 5.1. *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 $\mathtt{state}_t$ has the following form:*

The value $\mathtt{index} = t$. For $i = 1$ to $\mathtt{levels}$, the set S_i is of the following form.

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*

$$((w_1 = v_1^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j}), \texttt{open} = (k+1)\cdot 2^i - 1,$$
$$\texttt{closing} = (k+1)\cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r).$$

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

$$((w_1 = v_1^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j}), \texttt{open} = (k+1)\cdot 2^i - 1 + 2^{i-1},$$
$$\texttt{closing} = (k+1)\cdot 2^i - 1, \texttt{count} = 0).$$

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

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*

$$((w_1 = v_1^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j})\texttt{open} = (k+1)\cdot 2^i - 1 + 2^{i-1},$$
$$\texttt{closing} = (k+1)\cdot 2^i - 1, \texttt{count} = r - 2^{i-1}).$$

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

Finally, $\texttt{current} = (v_1^{\prod_{i\in[1,T]\setminus\{t\}} e_i}, \ldots, v_{\texttt{len}}^{\prod_{i\in[1,T]\setminus\{t\}} e_i})$.

We defer the proof of this lemma to Appendix A since it is rather involved, but follows mostly along the lines established in [27].

Claim 5.2. *The Update/Output Correctness condition of Definition 4.1 holds for our construction.*

Proof. This follows immediately from the statement of Lemma 5.1 regarding the form of $\texttt{current}$ at $\texttt{state}_t$. □

Claim 5.3. *The Shift Correctness condition of Definition 4.1 holds for our construction.*

Proof. Consider any $N \in \mathbb{Z}, T \in \mathbb{Z}, H : \{1 \ldots, T\} \to \mathbb{Z}, \texttt{len} \in \mathbb{Z}, (v_1, \ldots, v_{\texttt{len}}) \in \mathbb{Z}_N^{\texttt{len}}$ and $(z_1, \ldots, z_{\texttt{len}}) \in \mathbb{Z}^{\texttt{len}}$. Let $\texttt{state} = \textsf{SeqSetup}(N, 1^T, H, 1^{\texttt{len}}, (v_1, \ldots, v_{\texttt{len}}))$, let $v_1' = v_1^{z_1}, \ldots, v_{\texttt{len}}' = v_{\texttt{len}}^{z_{\texttt{len}}}$ and then let $\texttt{state}' = \textsf{SeqSetup}(N, 1^T, H, 1^{\texttt{len}}, (v_1', \ldots, v_{\texttt{len}}'))$. Finally, let $\widetilde{\texttt{state}} = \textsf{SeqShift}(\texttt{state}, (z_1, \ldots, z_{\texttt{len}}))$. We must verify that $\texttt{state}' = \widetilde{\texttt{state}}$.

We will check this for the case of $\texttt{index} = 1$ by going through each of the possible cases from Lemma 5.1 for each $i \in [1, \texttt{levels}]$. For $i \in [1, \texttt{levels}]$, let S_i denote the sets for $\texttt{state}$, S_i' be the sets for $\texttt{state}'$ and $\tilde{S}_i$ be the sets for $\widetilde{\texttt{state}}$.

Case 1: $T - \texttt{index} \leq 2^i - 2$. By Lemma 5.1, neither $\texttt{state}$ nor $\texttt{state}'$ has any elements in S_i or S_i'. The $\textsf{SeqShift}$ algorithm does not add any new elements to a level i, so $\tilde{S}_i$ is an empty set just like S_i'.

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

$$((w_1 = v_1^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j}), \texttt{open} = (k+1)\cdot 2^i - 1, \\ \texttt{closing} = (k+1)\cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r)$$

here we let $R = [\texttt{open}, \texttt{open} + 2^{i-1} - 1] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1} - 1]$. Applying the $\mathsf{SeqShift}$ operation produces

$$(w_1^{z_1} = (v_1^{\prod_{j\in[1,T]\setminus R} e_j})^{z_1} = (v_1')^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}}^{z_{\texttt{len}}} = (v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j})^{z_{\texttt{len}}} = (v_{\texttt{len}}')^{\prod_{j\in[1,T]\setminus R} e_j})$$

with the same $\texttt{open}, \texttt{closing}$ and $\texttt{count}$ values matching the corresponding element in S_i'.

The second is a tuple

$$((w_1 = v_1^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j}), \texttt{open} = (k+1)\cdot 2^i - 1 + 2^{i-1}, \\ \texttt{closing} = (k+1)\cdot 2^i - 1, \texttt{count} = 0).$$

where $R = [\texttt{open}, \texttt{open} + 2^{i-1} - 1] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1} - 1]$. Applying the $\mathsf{SeqShift}$ operation produces

$$w_1^{z_1} = (v_1^{\prod_{j\in[1,T]\setminus R} e_j})^{z_1} = (v_1')^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_1^{z_{\texttt{len}}} = (v_1^{\prod_{j\in[1,T]\setminus R} e_j})^{z_{\texttt{len}}} = (v_{\texttt{len}}')^{\prod_{j\in[1,T]\setminus R} e_j})$$

with the same $\texttt{open}, \texttt{closing}$ and $\texttt{count}$ values matching the corresponding element in S_i'.

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

$$((w_1 = v_1^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in[1,T]\setminus R} e_j}), \texttt{open} = (k+1)\cdot 2^i - 1 + 2^{i-1}, \\ \texttt{closing} = (k+1)\cdot 2^i - 1, \texttt{count} = r - 2^{i-1}).$$

where $R = [\texttt{open}, \texttt{open} + 2^{i-1}] \cup [\texttt{closing} + \texttt{count}, \texttt{closing} + 2^{i-1}]$. Applying the $\mathsf{SeqShift}$ operation gives

$$(w_1^{z_1} = (v_1^{\prod_{j\in[1,T]\setminus R} e_j})^{z_1} = (v_1')^{\prod_{j\in[1,T]\setminus R} e_j}, \ldots, w_{\texttt{len}}^{z_{\texttt{len}}} = (v_1^{\prod_{j\in[1,T]\setminus R} e_j})^{z_{\texttt{len}}} = (v_{\texttt{len}}')^{\prod_{j\in[1,T]\setminus R} e_j})$$

with the same $\texttt{open}, \texttt{closing}$ and $\texttt{count}$ values matching the corresponding element in S_i'.

Finally, $\texttt{current} = (v_1^{\prod_{i\in[1,T]\setminus\{t\}} e_i}, \ldots, v_{\texttt{len}}^{\prod_{i\in[1,T]\setminus\{t\}} e_i})$ and after the $\mathsf{SeqShift}$ operation we have

$$\tilde{\texttt{current}} = ((v_1^{\prod_{i\in[1,T]\setminus\{t\}} e_i})^{z_1} = (v_1')^{\prod_{i\in[1,T]\setminus\{t\}} e_i}, \ldots, (v_{\texttt{len}}^{\prod_{i\in[1,T]\setminus\{t\}} e_i})^{z_{\texttt{len}}} = (v_{\texttt{len}}')^{\prod_{i\in[1,T]\setminus\{t\}} e_i}$$

which again matches the $\texttt{state}'$.

□

5.3 The $\mathsf{SeqProgram}$ Algorithm

We conclude with describing the $\mathsf{SeqProgram}$ algorithm. We organize its description around Lemma 5.1 where for each S_i we determine what to add to it based on the three cases above. We then subsequently argue correctness in tandem.

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 $v' = v^{\prod_{i \in [1,\texttt{start}-1]} e_i}$. Therefore we must check (in the correctness argument) that in every case $R \cap [1, \texttt{start} - 1] = \emptyset$. If so, we can let $X = [1,T] \setminus (R \cup [1, \texttt{start} - 1])$ and compute $(v')^{\prod_{j \in X} e_j} = v^{\prod_{j \in [1,T] \setminus R} e_j}$.

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

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

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

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

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

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

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

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

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

Claim 5.4. *The Program correctness condition of Definition 4.1 holds for our construction.*

Proof. Our algorithm follows Lemma 5.1 exactly in terms of the number of tuples added in each case and their respective opening, closing and count values. Thus verifying correctness is a matter of verifying the formation of the $\texttt{len}$ group elements in each tuple added.

Suppose that $v'_1 = v_1^{\prod_{j \in [1,\texttt{start}-1]} e_j}, \ldots, v'_{\texttt{len}} = v_{\texttt{len}}^{\prod_{i \in [1,\texttt{start}-1]} e_j}$. It follows that for any $R \subseteq [1,T]$ if $R \cap [1, \texttt{start} - 1] = \emptyset$ and we let $X = [1,T] \setminus (R \cup [1, \texttt{start} - 1])$, then

$$(v'_1)^{\prod_{j \in X} e_j} = v_1^{\prod_{j \in [1,T] \setminus R} e_j}, \ldots, (v'_{\texttt{len}})^{\prod_{j \in X} e_j} = v_{\texttt{len}}^{\prod_{j \in [1,T] \setminus R} e_j}.$$

It then follows that all we must do is verify that $R \cap [1, \texttt{start} - 1] = \emptyset$ in every place the algorithm adds a tuple. Case 1 is trivial since no tuples are added so we focus on Cases 2 and 3.

Recall that $\texttt{index} = \texttt{start}$ so we will be able to use these terms interchangeably. In Case 2 we have $\texttt{index} = k \cdot 2^i + r$ for $0 \le r < 2^{i-1}$ which means that $\texttt{start} \le (k+1) \cdot 2^i - 1$ and that $(\texttt{start} - 1) < (k+1) \cdot 2^i - 1$. For the first tuple added R consists of a union of two integer ranges both of which begin with a value of at least $(k+1) \cdot 2^i - 1$ therefore the set $[1, \texttt{start} - 1]$ cannot intersect with this R. For the second tuple there is another R set, but again it consists of a union of two integer ranges both of which have starting values at least $(k+1) \cdot 2^i - 1$. So again it cannot intersect with $[1, \texttt{start} - 1]$.

For Case 3 the argument is very similar. Again we have that $(\texttt{start} - 1) < (k+1) \cdot 2^i - 1$. There is a single R to consider which is a union of two ranges both of which have a start value at least $(k+1) \cdot 2^i - 1$ and therefore cannot intersect with $[1, \texttt{start} - 1]$.

The correctness of setting $\texttt{current}$ follows from the fact that $X \cup [1, \texttt{start} - 1] = [1, T] \setminus \{\texttt{start}\}$. □

We briefly remark that all algorithms are polynomial time in the input. The concrete efficiency of the $\mathsf{SeqProgram}$ 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

We now describe our a random oracle scheme using RSA sequencers. While our eventual goal is to put forth a standard model scheme, exploring the system below is useful as it shows how to build a signature from the RSA sequencer abstraction as well as leverage the $\mathsf{SeqShift}$ algorithm to push the expensive cost of setting up the sequencer to a one time global setup process. The latter allows us to perform key generation very efficiently. Our resulting scheme has key generation and update times that scale logarithmically in T. It is close in form and performance to the Itkis-Reyzin scheme with the "pebbling" optimization applied. The important exception though is that the Itkis-Reyzin scheme has expensive key generation that grows linearly in T.

We describe the scheme below.

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 $G : \mathbb{Z}_N \times \{0,1\}^L \times [1, T] \to [0, 2^\lambda - 1]$ 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 [26] and slightly refined in [27] to map integers to primes of an appropriate size. This hash function will not require the random oracle heuristic. The hash function $H : [1, T] \to \{0,1\}^{\lambda+1}$ 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 $F : [1,T] \times [1, \lambda \cdot (\lambda^2 + \lambda)] \to \{0,1\}^\lambda$, 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}.[6] 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 $e_t = H_K(t)$.

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. We are now ready to describe the algorithms comprising our scheme.

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

$$\mathtt{state}_{\mathsf{pp}} = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=1}, g).^{7}$$

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 parses $\mathsf{pp} = (T, N, Y, K, \mathtt{state}_{\mathsf{pp}})$. It chooses a random integer u in $[1, N]$. It computes $\mathtt{state}_1 = \mathsf{SeqShift}(\mathtt{state}_{\mathsf{pp}}, u)$. Next it computes $U = Y^u \mod N$ 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 $s = \mathsf{SeqCurrent}(\mathtt{state}_t)$.[8] It next chooses a random $r \in \mathbb{Z}_N^*$ and computes $\sigma_2 = G(r^{e_t} \mod N, M, t)$. 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 $\sigma_1 = 0 \mod N$; 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

$$G(a, M, t) \overset{?}{=} \sigma_2.$$

[6]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.

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

[8]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).

6.2 Correctness

Consider a public/private key pair that was created with random exponent u and let $\mathtt{state}_t$ denote the secret key state on time period t (i.e., after $t-1$ updates). In addition, assume the global setup was created with modulus N and generator g.

By shift correctness of the sequencer, we have that $\mathtt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=1}, g^u)$. I.e. Even though $\mathtt{state}_1$ was computed from $\mathsf{SeqShift}(\mathtt{state}_{\mathsf{pp}}, u)$ it will have the same distribution as if it were initially setup with the element g^u.

Now consider a signature generated for time period t where $\mathtt{state}_t$ is the state after the key was updated $t-1$ times. When the signing algorithm calls $s = \mathsf{SeqCurrent}(\mathtt{state}_t)$, the update/output correctness of the sequencer guarantees that $s = (g^u)^{\prod_{i \in [1,T] \setminus \{t\}} e_i} = U^{1/e_t}$. The first equality follows from the definition of update/output correctness. The second comes from the definition of $U = g^u$ and the fact that e_t must be relatively prime to $\phi(N)$ by the way they are chosen.

Let $r \in \mathbb{Z}_N^*$ be the random element chosen for generating the signature and $\sigma_2 = G(r^{e_t}, M, t)$, then

$$\sigma_1 = r \cdot s^{\sigma_2} = r \cdot (g^{u/e_t})^{\sigma_2}.$$

The verification algorithm will compute:

$$a = \sigma_1^{e_t}/(U^{\sigma_2}) = (r \cdot (g^{u/e_t})^{\sigma_2})^{e_t}/(U^{\sigma_2}) = r^{e_t} \cdot (g^u)^{\sigma_2}/(U^{\sigma_2}) = r^{e_t}.$$

The verification will then accept since $G(a, M, t) = G(r^{e_t}, M, t) = \sigma_2$.

6.3 Efficiency

We will analyze the efficiency of our construction with regard to a general sequencer, but also make specific remarks regarding the efficiency when instantiated with the sequencer of Section 5. In Section 9, we will give a more detailed evaluation with concrete choices of security parameters.

The global Setup algorithm is dominated by one call to $\mathsf{SeqSetup}$ which involves approximately $T \cdot \lg(T)$ exponentiations and prime searches for our implementation. It also requires T modular multiplications in $\phi(N)$ and T prime searches when computing Y. The KeyGen operation requires approximately $2\lg(T)$ exponentiations. The Update algorithm is simply a call to $\mathsf{SeqUpdate}$ which will take $\lg(T)$ exponentiations and prime searches using the construction of Section 5. The signing algorithm makes a single call to $\mathsf{SeqCurrent}$, which in our construction is just a fetch of an element of $\mathbb{Z}_N^*$. The dominant signing costs are two exponentiations as well as the prime search for e_t. Finally verification is dominated by the cost of two exponentiations and a prime search.

We conclude with a few remarks on how efficiency can be improved.

- The global setup performs several prime searches in computing $\mathtt{state}_{\mathsf{pp}}$ and Y. One could amortize the costs by sharing the prime search across both implementations. This involves slightly breaking the modularity of the abstractions.
- As we described, the signature verification involves an inversion mod N. Alternatively, one could compute $Y = (g^E)^{-1} \mod N$ and save the inversion per signature. We kept our original description since we found it slightly more favorable pedagogically.
- When setting parameters in an implementation, the output length of G will depend on the maximum number of oracle queries (hash function evaluations) the adversary can make. On

the other hand the length of prime exponents will need to be sufficiently long to avoid a collision which will be impacted by the number of total time periods T. We generally expect the former to be larger than the latter, however; our proof relies on the exponent e_t values to be bigger than the outputs of G. One way to possibly avoid this inflation is to first search for smaller value primes and then use a small power of these primes as the e_t value. For example, search for a smaller e'_t prime, but use $e_t = (e'_t)^c$ such that e_t is still guaranteed to be larger than the maximum value output by G. Doing this would require adjusting our security proof to handle powers of primes. This nice idea of using powers of smaller primes appeared in Itkis-Reyzin [28].

- The KeyGen, Update, Sign and Verify algorithms all received the public parameters pp; however, not all parts of the parameters are needed by each algorithm. For example, only KeyGen requires $\mathtt{state}_{\mathsf{pp}}$ which is the largest component of the parameters. A party that only verifies might want to just store N and K along with the individual public keys.

6.4 Proof of Security

Theorem 6.1. *If the RSA assumption (Assumption 3.1) 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 2.2.*

Proof. The reduction algorithm receives an RSA challenge (N, e^*, h) and needs to use the attacker to compute $h^{1/e^*} \mod N$. In this proof, the function $G : \mathbb{Z}_N \times \{0,1\}^L \times [1,T] \to [0, 2^\lambda - 1]$, where N is an RSA modulus output from the global setup, is modeled as a random oracle. We establish some syntax regarding this for future use. We parse the queries of the form made to the random oracle into two types. The first we call "Type A" queries ones where the attacker makes the query to G itself. For "Type B" queries, these are made by the challenger during a call to a Signing query. Let $Q_{A,t}$ be the maximum number of Type A queries made for time period t; that is, inputs of the form $(\cdot,\cdot,t)$. Each input of the form (a, m, t) will only be counted once and be labeled by how it was *first* called. Also, the forgery verification will induce a random oracle query; if this input has not already been queried, then count this as the last Type A query for the forgery period.[9]

We now define a sequence of games.

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

Game 2: (Guessing the forgery period) The same as Game 1, except at the start the game randomly guesses a time period $t^* \in [1, T]$ that the attacker forges on. The adversary only wins if he would have won Game 1 and the guess of t^* was correct.

Game 3: (Guessing the critical RO query) The same as Game 2, except at the start the game randomly guesses an index $q^* \in [1, Q_{A,t^*}]$ that corresponds to the q^*th query to the random oracle by the attacker for time period t^*. Let that query be of the form $(a_{q^*}, m_{q^*}, t^*_{q^*} = t^*)$. The adversary only wins if he would have won Game 2 and m_{q^*} is the message that the attacker forges on.

[9]Without loss of generality, we can assume that the attacker makes a random oracle query on the input used in the forgery, if it was not already queried. If it does not, we can trivially create an adversary that is a pass through between the challenger and the attacker, except that on forgery output $(m, t, (\sigma_1, \sigma_2))$ by the attacker, it first queries the random oracle on input $(\sigma_1^{e_t}/U^{\sigma_2}, m, t)$ before passing the forgery output to the challenger.

Game 4: (Forking at the critical RO query) The same as Game 3, except the challenger proceeds in the following way:

- The challenger simulates Game 2 for the attacker up until the q^*th Type A query to the random oracle for time period t^*. Let this input be (a^*, m^*, t^*). At this point, it saves the state of the game including the state of the attacker.
- The challenger then returns a random value in $[0, 2^\lambda - 1]$ as the random oracle's response to this query and then runs the rest of the game with the attacker. At the end of the game, it records the attacker's output.
- The challenger then reloads the saved information about the state of the game and the attacker's state at the time it issued the query (a^*, m^*, t^*) to the random oracle. The challenger chooses a second random response in $[0, 2^\lambda - 1]$ and then finishes Game 2 with the attacker, recording this output as well.
- The attacker wins in this game only if the attacker won in *both* forks of the game. (We note that the condition of forging on the same message m^* and period t^* in both forks is already inherited from Games 3 and 2, respectively, above.)

Game 5: (H_K does not default) The attacker wins only if it meets all the conditions to win in Game 4 and $H_K(t^*) \neq e_{\text{default}}$ (that is, the default condition of the hash is not triggered on the forgery period or otherwise equal to the default prime.)

Game 6: (H_K does not collide) The attacker wins only if it meets all the conditions to win in Game 5 and $H_K(t^*) \neq H_K(t)$ for all $t \in [1, T]$ where $t \neq t^*$.

Game 7: (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 6 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 8: (Programming H_K with random value) The same as Game 7, except that it chooses a random $y' \in \{0,1\}^\lambda$ and sets $c = y' \oplus F_{K'}(t^*, i^*)$.

Game 9: (Programming H_K with e^*) The same as Game 8, 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 $c = y' \oplus F_{K'}(t^*, i^*)$.

Game 10: (Key Generation with SeqSetup) The same as Game 9, except when generating $\texttt{state}_1$ as part of sk_1, the challenger computes this as $\texttt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\texttt{len}=1}, g^u)$ (instead of $\texttt{state}_1 = \mathsf{SeqShift}(\texttt{state}_{\mathsf{pp}}, u)$).

Game 11: (Key Generation with Random Quadratic Residue) The same as Game 10, except during key generation, the challenger chooses a random $w \in \text{QR}_N$ and then sets $\texttt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\texttt{len}=1}, w)$ and $U = w^{\prod_{i \in [1,T]} e_i}$.

Game 12: (Handling the Break-In) The same as Game 11, except that when the attacker issues a break-in query for time period b, if $b \leq t^*$, then the challenger immediately aborts and the attacker does not win. Otherwise, if $b > t^*$, then the challenger returns $\mathsf{sk}_b = (\texttt{state}_b, e_b, b)$, where $\texttt{state}_b$ is computed as $\texttt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\texttt{len}=1}, w^{\prod_{i \in [1,b-1]} e_i}, b)$.

Game 13: (Signing for Periods $t \neq t^*$) The same as Game 12, except when the attacker asks for a signature query on (m, t), where $t \neq t^*$, instead of computing s as $s = \mathsf{SeqCurrent}(\mathtt{state}_t)$, compute it as $s = w^{\prod_{i \in [1,T] \setminus \{t\}} e_i}$. The signature algorithm then proceeds in the same way as before.

Game 14: (Preparing to embed the RSA Challenge h) The same as Game 13, except as follows. The challenger lets $h = w^{e^*} = w^{e_{t^*}}$ for some w. It then sets the public key information $U = h^{\prod_{i \in [1,T] \setminus \{t^*\}} e_i} = w^{\prod_{i \in [1,T]} e_i}$. When the attacker asks for a signature query on (m, t), where $t \neq t^*$, the challenger computes $s = h^{\prod_{i \in [1,T] \setminus \{t, t^*\}} e_i} = w^{\prod_{i \in [1,T] \setminus \{t\}} e_i}$. The secret key for a break-in at time b is computed as $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$, where $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=1}, h^{\prod_{i \in [1,b-1] \setminus \{t^*\}} e_i}, b)$. Looking ahead to our main reduction, we will use h as the RSA challenge and w as its solution.

Game 15: (Programming the Random Oracle) The same as Game 14, except we change how the challenger responds to signature queries for time period t^*. On signature query (m, t^*), the challenger chooses a random $\sigma_2 \in [0, 2^\lambda - 1]$ and a random $r' \in \mathbb{Z}_N^*$. It computes $a = r'^{e_{t^*}} \cdot U^{-\sigma_2}$, sets $\sigma_1 = r'$ and programs the random oracle to respond so that $\sigma_2 = G(a, m, t^*)$. If the challenger has already given out a different response for G on this input (a, m, t^*), then the challenger aborts immediately and the attacker does not win. Otherwise, the challenger responds with $\sigma = (\sigma_1, \sigma_2)$.

Next, we establish a series of claims that show that if an adversary is successful in the real forward security game (Game 1) then it will be successful in Game 15 as well. We then describe a simulator that can use any adversary successful in Game 15 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 6.2.

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

Proof. The adversary's view in Games 1 and 2 is identical. The only difference is whether or not the guess of the forgery period $t^* \in [1, T]$ is correct. This occurs with probability $1/T$. □

Claim 6.3.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 3}] \geq \frac{\mathbf{Adv}_{\mathcal{A}}[\text{Game 2}]}{Q_{A,t^*}}.$$

Proof. The adversary's view in Games 2 and 3 is identical. The only difference is whether or not the guess of $q^* \in [1, Q_{A,t^*}]$ is correct. This occurs with probability $1/Q_{A,t^*}$. □

Claim 6.4.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 4}] = (\mathbf{Adv}_{\mathcal{A}}[\text{Game 3}])^2.$$

Proof. The adversary's view in Games 3 and 4 is identical, except that the simulation forks at the random oracle query $(a_{q^*}, m_{q^*}, t_{q^*})$, records the results of the two simulations, and the attacker only wins in Game 4 if she wins in both simulations. We note that in the (forked) second simulation, the challenger simply chooses a fresh random value in $[0, 2^\lambda - 1]$ (possibly equal to the response of the first simulation), so the adversary's view and the distribution of the challenger's response is the same. Thus, due to the Forking Lemma [37], the probability of the attacker winning in both forks is the square of the probability of her winning in a single run. □

Claim 6.5. *If F is a secure pseudorandom function and $\lambda \geq 4$, then*

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 5}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 4}] - \mathsf{negl}(\lambda).$$

Claim 6.6. *If F is a secure pseudorandom function and $T \in \mathsf{poly}(\lambda)$, then*

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 6}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 5}] - \mathsf{negl}(\lambda).$$

Claim 6.7.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 7}] = \frac{\mathbf{Adv}_{\mathcal{A}}[\text{Game 6}]}{\lambda^3 + \lambda^2}.$$

Claim 6.8.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 8}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 7}].$$

Claim 6.9.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 9}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 8}].$$

The proofs of Claims 6.5-6.9 appear in Appendix B and are similar to those used in [27] to program the hash function H_K with a desired RSA exponent. We make only minor modifications to address the fork that occurred in Game 3.

Claim 6.10.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 10}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 9}].$$

Proof. The attacker's view in these games is identical. The only difference is how the $\mathtt{state}_1$ value is computed, but both computations result in the same output and this follows from the correctness of $\mathsf{SeqShift}$. □

Claim 6.11.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 11}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 10}] - \mathsf{negl}(\lambda).$$

Proof. The only difference between the attacker's view in these games is the distribution of the public and secret key, which we argue are statistically close. In Game 10, the base for $\mathsf{SeqSetup}$ (which generates sk_1) and the base for U which forms the pk are both g^u, where g is a generator of QR_N and u is chosen randomly from $[1, N]$. In Game 11, g^u is replaced by a random $w \in \mathrm{QR}_N$, which is equivalent to having $w = g^{u'}$ with $u' \in [1, \phi(N)]$. Since $N = \phi(N) + p + q + 1$, where the difference between these two ranges of $(p + q + 1)$ is negligible, the attacker will not be able to distinguish. □

Claim 6.12.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 12}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 11}].$$

Proof. We argue in two cases that an attacker's advantage in these games is identical. First, in Game 12, when the break-in period $b \leq t^*$, then the challenger aborts and the attacker automatically loses. However, the same thing happens in Game 11. Due to Game 2, we know that the attacker only wins if the forgery period is t^*. Due to Game 1 (forward security game), we know that the attacker only wins if she forges for a period prior to the break-in. Thus, in this case, there is no way for the attacker to win, so aborting immediately and declaring the loss will not change anything.

Second, in Game 12, when $b > t^*$, the only change is how the secret key $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$ that the challenger must respond with is computed. In Game 12, $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=1},$

$w^{\prod_{i\in[1,b-1]} e_i}, b)$ whereas in Game 11, $\texttt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\texttt{len}=1}, w)$ and then $\texttt{state}_{t+1} = \mathsf{SeqUpdate}(\texttt{state}_t)$ for $t = 1$ to $b-1$. By the correctness of $\mathsf{SeqProgram}$, these two methods of computing $\texttt{state}_b$ will result in the same output. Thus, there is nothing for the adversary to distinguish in this case. □

Claim 6.13.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 13}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 12}].$$

Proof. An attacker's advantage in these two games is the same. The value s is the same value in both games, the challenger just computes the same thing in two different ways. This follows from the correctness of $\mathsf{SeqSetup}$ and $\mathsf{SeqUpdate}$. □

Claim 6.14.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 14}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 13}].$$

Proof. An attacker's advantage in these two games is the same. Per Assumption 3.1, $h = w^{e*} = w^{e_{t^*}}$ mod N for a random $w \in \mathrm{QR}_N$, so the distribution for U and s is identical. At the time of the break in, both games inherit from Game 12 that $b > t^*$ (otherwise, the challenger immediately aborts), so it is possible to compute those identical inputs to $\mathsf{SeqProgram}$ using h; namely that, $h^{\prod_{i\in[1,b-1]\setminus\{t^*\}} e_i} = w^{\prod_{i\in[1,b]} e_i}$. □

Claim 6.15.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 15}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 14}] - \mathsf{negl}(\lambda).$$

Proof. Our argument will proceed as follows. We first observe that the challenger's signature response in Game 15 will verify. We then argue that the distribution of these signatures in Game 15 is the same as the distribution of the corresponding signatures of Game 14. We conclude by arguing that the probability that the challenger is forced to abort in Game 15 due to a collision with a prior response of G is negligible.

First, we show that the challenger's signature responses are correct. On input (m, t^*), the challenger responds with $\sigma = (\sigma_1, \sigma_2)$. The verification algorithm computes $a = \sigma_1^{e_{t^*}} \cdot U^{-\sigma_2}$ and queries $G(a, m, t^*)$, which the challenger programmed to return σ_2 on this query. The verification algorithm then outputs 1 because $G(a, m, t^*) = \sigma_2$.

Second, we look at the distribution of the signature $\sigma = (\sigma_1, \sigma_2)$. In Game 15, σ_2 was chosen by the challenger to be a random value in the range $[0, 2^\lambda - 1]$, which is identical to how this value is chosen by the challenger in Game 14. In Game 15, σ_1 is chosen as a random value in $\mathbb{Z}_N^*$, where in Game 14 it is computed as $\sigma_1 = r \cdot s^{\sigma_2}$, where r is a random value in $\mathbb{Z}_N^*$; thus, they have the same distribution.

Finally, in Game 15, the challenger computes $a = r'^{e_{t^*}} \cdot U^{-\sigma_2}$ and aborts if it has already issued a random oracle G response on input (a, m, t^*) that does not equal σ_2. The probability of the challenger aborting in this scenario is the probability that it previously selected an $r \in \mathbb{Z}_N^*$ randomly such that $a = r^{e_{t^*}} \mod N$, which is negligible. □

6.4.1 Main Reduction

We now show that if there exists a polynomial-time (in λ) attacker that has advantage $\epsilon = \epsilon(\lambda)$ in Game 15, then there exists a polynomial-time (in λ) attacker for the RSA problem in Assumption 3.1 with advantage $\epsilon - \mathsf{negl}(\lambda)$.

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

Setup.

1. Obtain 1^T from the signature adversary $\mathcal{A}$.
2. Make random guesses of $t^* \in [1, T], q^* \in [1, Q_{A,t^*}], i^* \in [1, \lambda^3 + \lambda^2]$.
3. Following [27] to program H_K, 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 $K = (K', c, e_{\text{default}})$. 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. Set $\mathtt{len} = 1$. Choose a random $g \in \mathrm{QR}_N$ and obtain $\mathtt{state}_{\mathsf{pp}} = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=1}, g)$.
5. Compute $Y = g^{\prod_{i=1}^{T} e_i} \mod N$.
6. Compute $U = h^{\prod_{i \in [1,T] \setminus \{t^*\}} e_i} \mod N$.
7. Set the $\mathsf{pp} = (T, N, Y, K, \mathtt{state}_{\mathsf{pp}})$ and $\mathsf{pk} = U$. Send $(\mathsf{pp}, \mathsf{pk})$ to $\mathcal{A}$.

Queries. The adversary can make three different types of queries.

1. Random Oracle G: The function $G : \mathbb{Z}_N \times \{0,1\}^L \times [1, T] \to [0, 2^\lambda - 1]$ will be under the challenger's control. On input (a, m, t), check to see if this input was previously queried to G (by the adversary or the challenger) or set by the challenger during a Signing query and if so, return that response. If not, chose a random value in $[0, 2^\lambda - 1]$, record this, and return it.

2. Signing: On input (t, m) from the adversary, if $t \neq t^*$, compute $s = h^{\prod_{i \in [1,T] \setminus \{t, t^*\}}} = w^{\prod_{i \in [1,T] \setminus \{t\}}}$ and then follow the rest of the signing algorithm from the construction and return that signature. This step may invoke calling G, which the challenger can simulate per the above. If $t = t^*$, ,then the challenger chooses a random $\sigma_2 \in [0, 2^\lambda - 1]$ and a random $r' \in \mathbb{Z}_N^*$. It computes $a = r'^{e_{t^*}} \cdot U^{-\sigma_2}$, sets $\sigma_1 = r'$ and programs the random oracle to respond so that $\sigma_2 = G(a, m, t^*)$. If the challenger has already given out a different response for G on this input (a, m, t^*), then the challenger aborts immediately. Otherwise, the challenger responds with $\sigma = (\sigma_1, \sigma_2)$.

3. Break-In: On input a break-in key request for period b from the adversary, if $b \leq t^*$, then the challenger aborts. If $b > t^*$, then the challenger returns $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$, where $\mathtt{state}_b$ is computed as $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=1}, h^{\prod_{i \in [1,b-1] \setminus \{t^*\}} e_i}, b)$.

Output. Eventually, due to the fork in Game 3, we obtain from the adversary two tuples (m, t, σ) and (m, t, σ'). If either signature tuple does not verify, if any of the challenger's guesses of t^*, q^*, i^* were incorrect, if t is at or after the break-in or the message-period pair of (m, t) was previously queried to the signing oracle, then abort. These abort conditions are all consistent with the adversary not winning Game 15. Otherwise, let $E' = \prod_{i\in[1,T]\setminus\{t^*\}} e_i$. Then, due to the fork in Game 3 at G on input (a, m, t^*), we have that $\sigma = (\sigma_1, \sigma_2)$ and $\sigma' = (\sigma_1', \sigma_2')$ such that

$$a = \sigma_1^{e^*} \cdot U^{-\sigma_2} = \sigma_1'^{e^*} \cdot U^{-\sigma_2}$$

If $\sigma_2 = \sigma_2'$, the challenger aborts. This happens with at most $1/2^\lambda$ probability, since these values are both chosen randomly from $[0, 2^\lambda - 1]$ in Game 3. Without loss of generality let $\sigma_2' > \sigma_2$, we have that

$$(\frac{\sigma_1}{\sigma_1'})^{e^*} = U^{-\sigma_2+\sigma_2'} = h^{\prod_{i\in[1,T]\setminus\{t^*\}} e_i \cdot (-\sigma_2+\sigma_2)}.$$

Next, we want to apply Lemma 3.2 to the above, where we have $x = \frac{\sigma_1}{\sigma_1'}$, $y = h$, $a' = e^*$ and $b' = \prod_{i\in[1,T]\setminus\{t^*\}} e_i \cdot (-\sigma_2 + \sigma_2)$, so we have an equation of the form $x^{a'} = y^{b'}$. Since we will argue that $\gcd(a', b') = 1$, we can apply the efficient algorithm of Cramer and Shoup for computing z such that $z^{e^*} = h$, which we then output as the solution to the RSA challenge.

Analysis. The attacker's view in the main reduction is the same as Game 15, except that the challenger will abort if $\sigma_2 = \sigma_2'$. As stated above this happens with $1/2^\lambda$ probability, since these values are both chosen randomly from $[0, 2^\lambda - 1]$ by the challenger in Game 3. Thus, the challenger will abort due to this reason with at most negligible probability. Also, in this reduction, the challenger uses the RSA challenge value h to compute the public key and related values, and that this challenge h has the same distribution as the h of Game 15 (both are derived via raising a random element of QR_N to e^*).

It remains to argue that $\gcd(a', b') = 1$, so that the challenger will be able to compute an RSA solution. Recall that $a' = e^*$ and $b' = \prod_{i\in[1,T]\setminus\{t^*\}} e_i \cdot (-\sigma_2 + \sigma_2)$. From Games 5 and 6, we know that the RSA challenge exponent e^* is a distinct prime from all other $e_i = H_K(i)$ for inputs $i \in [1, T]$. So e^* is relatively prime to $\prod_{i\in[1,T]\setminus\{t^*\}} e_i$. Since $\sigma_2' - \sigma_2$ is in the range $[0, 2^\lambda - 1]$ and e^* is a prime chosen from the range 2^λ to $2^{\lambda+1} - 1$, then we can conclude that e^* is relatively prime to $(-\sigma_2 + \sigma_2)$. Thus, $\gcd(a', b') = 1$ as needed.

□

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.

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 9, 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 first parses the $L = (\ell k)$-bit message M as $M = m_1|m_2|\ldots|m_k$, 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

$$\sigma_1^{e_t} \stackrel{?}{=} U_0 \cdot \tilde{U}^{\sigma_2} \cdot \prod_{j=1}^{k} U_j^{m_j}.$$

7.2 Correctness

Consider public parameters created via $\mathsf{SeqSetup}$ with generator g and modulus N, and from this a key pair $(\mathsf{pk}, \mathsf{sk}_1)$ created via $\mathsf{SeqShift}$ with random exponents $(u_0, u_1, \ldots, u_k, \tilde{u})$. At time period $t \in [1, T]$ (i.e., after $t-1$ updates), by the correctness of $\mathsf{SeqUpdate}$ and $\mathsf{SeqCurrent}$, we have that $s_j = (g^{u_j})^{\prod_{i \in [1,T] \setminus \{t\}} e_i} = U_j^{1/e_t}$ for $j \in [0, k]$ and $\tilde{s} = (g^{\tilde{u}})^{\prod_{i \in [1,T] \setminus \{t\}} e_i} = \tilde{U}^{1/e_t}$.

Let $r \in [0, 2^\lambda - f(\lambda)]$ be the random integer chosen in generating the signature, then

$$\sigma_1 = s_0 \cdot \tilde{s}^r \cdot \prod_{j=1}^{k} s_j^{m_j} = g^{u_0 E/e_t} \cdot (g^{\tilde{u}E/e_t})^r \cdot \prod_{j=1}^{k} (g^{u_j E/e_t})^{m_j}.$$

When the verification algorithm raises σ_1 to the e_t, it will get the needed equivalence of

$$g^{u_0 E} \cdot (g^{\tilde{u}E})^r \cdot \prod_{j=1}^{k} (g^{u_j E})^{m_j} = U_0 \cdot \tilde{U}^{\sigma_2} \cdot \prod_{j=1}^{k} U_j^{m_j}.$$

7.3 Efficiency

The efficiency of this construction will follow somewhat similarly to the analysis of Section 6.3. One notable difference is that the sequencer will need to keep track of $k+2$ values instead of 1. For that reason it is generally preferable to use smaller k values and choosing $k = 1$ is a natural choice. However, one advantage of using slightly large k values is that it allows for smaller sized message chunks which in turn enable smaller prime e_t exponents which must be larger than the message chunk size. We will explore different concrete options in our Section 9 evaluation.

The global Setup algorithm is dominated by one call to $\mathsf{SeqSetup}$ which involves approximately T prime searches for our implementation (via the optimization in Section 5.1). The canonical implementation calls for $(k+2)T \cdot \lg(T)$ exponentiations; however, one can observe that since $(v_1 = g, v_2 = g, \ldots, v_{\mathtt{len}} = g)$ the sequencer is basically computing the exact same thing $k+2$ times. And by breaking the abstraction slightly one can compute this with approximately $T \cdot \lg(T)$ exponentiations. The KeyGen operation requires approximately $2(k+2)\lg(T)$ exponentiations. The Update algorithm is simply a call to $\mathsf{SeqUpdate}$ which will take $(k+2)\lg(T)$ exponentiations and $\lg(T)$ prime searches using the construction of Section 5. The signing algorithm makes a single call to $\mathsf{SeqCurrent}$, which in our construction is just a fetch of a tuple of elements in $\mathbb{Z}_N^*$. The dominant costs are the exponentiation to r of approximately λ bits and k exponentiations to ℓ bit values as well as the prime search for e_t. Finally verification will be dominated by the exponentiation to r of approximately λ bits and k exponentiations to ℓ bit values and a prime search.

7.4 Proof of Security

Theorem 7.1. *If the RSA assumption (Assumption 3.1) holds and F is a secure pseudorandom function, then the Section 7.1 key-evolving signature construction is single-sign forward secure.*

Proof. The reduction algorithm receives an RSA challenge (N, e^*, h) and needs to use the attacker to compute $h^{1/e^*} \mod N$. We utilize a sequence of games which are similar in structure, but with a number of critical technical differences, to those of Section 6.4. We will assume that the attacker asks for a signing query (on any message of its choice) during time period t^*, which is the forgery period used by the attacker. This is without loss of generality, because given an attacker that does not (always) make a query during this period, we can simulate an attacker that does by issuing a signing query during t^* on a random message if the adversary fails to do so, before outputting the final forgery.

We now define a sequence of games.

Game 1: (Security Game) This game is defined to be the same as the *single sign* forward security game.

Game 2: (Guessing the forgery period and message chunk) The same as Game 1, except the game randomly guesses a time period $t^* \in [1, T]$ that the attacker forges on and a message chunk $\alpha \in [1, k]$ that will differ between the attacker's signing query at t^* and the forgery message. Let $M = m_1|m_2|\dots|m_k$ and $M^* = m_1^*|m_2^*|\dots|m_k^*$ be the messages of the signing query at t^* and the forgery output, respectively. The adversary only wins if he would have won Game 1 and the guess of t^* was correct and $m_\alpha \neq m_\alpha^*$.

Game 3: (Signing on t^* with special r) The same as Game 2, except that the challenger chooses random values $d \in [0, 2^\lambda - f(\lambda)]$ and $\beta \in \{0, 1\}$. When the attacker makes a signing query of the form $(M = m_1|\dots|m_k, t)$, if $t \neq t^*$, then the challenger signs the message following the regular signing algorithm. If $t = t^*$, then the adversary sets $r = d + \beta \cdot m_\alpha$ (instead of choosing it randomly) and then from this point follows the regular signing algorithm.

Game 4: (Aborting on collision with attacker) The same as Game 3, except the attacker only wins with forgery output $(M = m_1|\dots|m_k, t, \sigma = (\sigma_1, \sigma_2))$ if he would have won in Game 3 and $\sigma_2 \neq d + \beta \cdot m_\alpha$ (otherwise the challenger aborts).

Game 5: (H_K does not default) The attacker wins only if it meets all the conditions to win in Game 4 and $H_K(t^*) \neq e_{\text{default}}$ (that is, the default condition of the hash is not triggered on the forgery period or otherwise equal to the default prime.)

Game 6: (H_K does not collide) The attacker wins only if it meets all the conditions to win in Game 5 and $H_K(t^*) \neq H_K(t)$ for all $t \in [1, T]$ where $t \neq t^*$.

Game 7: (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 6 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 8: (Programming H_K with random value) The same as Game 7, except that it chooses a random $y' \in \{0, 1\}^\lambda$ and sets $c = y' \oplus F_{K'}(t^*, i^*)$.

Game 9: (Programming H_K with e^*) The same as Game 8, 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 $c = y' \oplus F_{K'}(t^*, i^*)$.

Game 10: (Key Generation with SeqSetup) The same as Game 9, except when generating $\mathtt{state}_1$ as part of sk_1, the challenger computes this as $\mathtt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (g^{u_0}, g^{u_1}, \ldots, g^{u_k}, g^{\tilde{u}}))$ (instead of $\mathtt{state}_1 = \mathsf{SeqShift}(\mathtt{state}_{\mathsf{pp}}, (u_0, u_1, \ldots, u_k, \tilde{u})))$.

Game 11: (Key Generation with Random Quadratic Residue) The same as Game 10, except during key generation, the challenger chooses a random $w \in \mathrm{QR}_N$. It computes $v_0 = g^{u_0} \cdot w^d$, $v_\alpha = g^{u_\alpha} \cdot w^\beta$, $\tilde{v} = g^{\tilde{u}} \cdot w^{-1}$ and for $i = 1$ to k, where $i \neq \alpha$, $v_i = g^{u_i}$. It then computes for $i = 0$ to k, $U_i = v_i^E$ and $\tilde{U} = \tilde{v}^E$, where $E = \prod_{i \in [1,T]} e_i$. It then sets $\mathtt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_0, \ldots, v_k, \tilde{v}))$ when computing $\mathsf{sk}_1 = (\mathtt{state}_1, e_1, 1)$ and then sets the public key as $\mathsf{pk} = (U_0, \ldots, U_k, \tilde{U})$.

Game 12: (Handling the Break-In) The same as Game 11, except that when the attacker issues a break-in query for time period b, if $b \leq t^*$, then the challenger immediately aborts and the attacker does not win. Otherwise, if $b > t^*$, then the challenger computes $\gamma = \prod_{i \in [1,b-1]} e_i$ and as in Game 11, $v_0 = g^{u_0} \cdot w^d$, $v_\alpha = g^{u_\alpha} \cdot w^\beta$, $\tilde{v} = g^{\tilde{u}} \cdot w^{-1}$ and for $i = 1$ to k, where $i \neq \alpha$, $v_i = g^{u_i}$. It returns $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$, where $\mathtt{state}_b$ is computed as $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_0^\gamma, \ldots, v_k^\gamma, \tilde{v}^\gamma), b)$.

Game 13: (Signing for Periods $t \neq t^*$) The same as Game 12, except when the attacker issues a signature query on $(M = m_1|\ldots|m_k, t)$, where $t \neq t^*$, instead of computing $(s_0, \ldots, s_k, \tilde{s})$ as $\mathsf{SeqCurrent}(\mathtt{state}_t)$, the challenger computes this tuple as follows. Let $\gamma = \prod_{i \in [1,T] \setminus \{t\}} e_i$. The challenger computes $s_0 = g^{\gamma u_0} \cdot w^{\gamma d}$, $s_\alpha = g^{\gamma u_\alpha} \cdot w^{\gamma \beta}$, $\tilde{s} = g^{\gamma \tilde{u}} \cdot w^{-\gamma}$ and for $i = 1$ to k, where $i \neq \alpha$, $s_i = g^{\gamma u_i}$. The signature algorithm then proceeds in the same way as before.

Game 14: (Preparing to embed the RSA Challenge h) The same as Game 13, except as follows. The challenger lets $h = w^{e^*} = w^{e_{t^*}}$ for some w. Looking ahead to our main reduction, we will use h as the RSA challenge and w as its solution. Let $E = \prod_{i \in [1,T]} e_i$ and $E' = \prod_{i \in [1,T] \setminus \{t^*\}} e_i$. It computes $U_0 = g^{Eu_0} \cdot h^{E'd}$, $U_\alpha = g^{Eu_\alpha} \cdot h^{E'\beta}$, $\tilde{U} = g^{E\tilde{u}} \cdot h^{-E'}$ and for $i = 1$ to k, where $i \neq \alpha$, $U_i = g^{Eu_i}$. It then sets the public key information $\mathsf{pk} = (U_0, \ldots, U_k, \tilde{U})$.

When the attacker asks for a signature query on (m, t), where $t \neq t^*$, let $\gamma' = \prod_{i \in [1,T] \setminus \{t, t^*\}} e_i$ and $\gamma = \gamma' \cdot e^*$. The challenger computes $s_0 = g^{\gamma u_0} \cdot h^{\gamma' d}$, $s_\alpha = g^{\gamma u_\alpha} \cdot h^{\gamma' \beta}$, $\tilde{s} = g^{\gamma \tilde{u}} \cdot h^{-\gamma'}$ and for $i = 1$ to k, where $i \neq \alpha$, $s_i = g^{\gamma u_i}$. The signature algorithm then proceeds in the same way as before.

The secret key for a break-in at time b is computed as $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$, where $\mathtt{state}_b$ is computed as follows. The challenger computes $\pi' = \prod_{i \in [1,b-1] \setminus \{t^*\}} e_i$ and $\pi = \pi' \cdot e^*$ and then $v_0 = g^{\pi u_0} \cdot h^{\pi' d}$, $v_\alpha = g^{\pi u_\alpha} \cdot h^{\pi' \beta}$, $\tilde{v} = g^{\pi \tilde{u}} \cdot h^{-\pi'}$ and for $i = 1$ to k, where $i \neq \alpha$, $v_i = g^{\pi u_i}$. It returns $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$, where $\mathtt{state}_b$ is computed as $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_0, \ldots, v_k, \tilde{v}), b)$.

Game 15: (Signing for Period t^*) The same as Game 14, except we change how the challenger responds to the signature query for time period t^*. Recall that there will only be one. On signature query $(M = m_1|\ldots|m_k, t^*)$, the challenger sets $r = d + \beta \cdot m_\alpha$ (per Game 3) and then computes the signature $\sigma = (\sigma_1, \sigma_2)$ as $\sigma_2 = r$ and

$$\sigma_1 \quad = \quad (g^{E' u_0}) \cdot (g^{E' \tilde{u}})^r \cdot \prod_{i=1}^{k} (g^{E' u_i m_i}).$$

Next, we establish a series of claims that show that if an adversary is successful in the real *single sign* forward security game (Game 1) then it will be successful in Game 15 as well. We then describe a simulator that can use any adversary successful in Game 15 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 7.2.

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

Proof. The adversary's view in Games 1 and 2 is identical. The only difference is whether or not the challenger's guess of the forgery period $t^* \in [1, T]$ and the message chuck $\alpha \in [1, k]$ are correct. This occurs with probability $1/(Tk)$. □

Claim 7.3.

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

Proof. The adversary's view in Games 2 and 3 differ in the distribution of the responses to the signature query to sign $M = m_1|\ldots|m_k$ at period t^*. In Game 2, the challenger chooses a random $r \in [0, 2^\lambda - f(\lambda)]$ and uses that to compute the signature. In Game 3, the challenger chooses a random $d \in [0, 2^\lambda - f(\lambda)]$, a random $\beta \in \{0, 1\}$ and sets $r = d + \beta \cdot m_\alpha$, which it then uses to compute the signature. (Recall that α is chosen in Game 2.) When $\beta = 0$, the two distributions for r are the same. When $\beta = 1$, then in Game 3, r is now set as the adversary choosing a value in $[0, 2^{f(\lambda)} - 1]$ (recall that message chunks are ℓ bits and $2^\ell \leq f(\lambda)$)) and then the challenger adds a random value in $[0, 2^\lambda - f(\lambda)]$. So for any choice of m_α, since $f(\lambda)/2^\lambda$ is negligible, these two distributions for r will be statistically close. Thus, the attacker distinguishes these games with at most negligible advantage. □

Claim 7.4.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 4}] = \frac{\mathbf{Adv}_{\mathcal{A}}[\text{Game 3}]}{2} - \mathsf{negl}(\lambda).$$

Proof. The adversary's view in Games 3 and 4 is the same, except if the adversary outputs a forgery tuple $(M^* = m_1^*|\ldots|m_k^*, t^*, \sigma^* = (\sigma_1^*, \sigma_2^*))$ where $\sigma_2^* = d + \beta \cdot m_\alpha^*$. In this event, the challenger in Game 3 proceeds, but the challenger in Game 4 aborts and the adversary loses. Let $M = m_1|\ldots|m_k$ be the message that the attacker requested a signature on for period t^* and let $\sigma = (\sigma_1, \sigma_2)$ be the challenger's response to that query. (Recall that we assumed w.l..o.g. that all attackers would query at period t^* and, by single sign, there will be only one query.) Then we know that in both games $\sigma_2 = d + \beta \cdot m_\alpha$. For all m_α, define good_{m_α} to be the set of integers $r \in [0, 2^\lambda - f(\lambda)]$ such that there exist integers $x, x' \in [0, 2^\lambda - f(\lambda)]$ such that $r = x + 0 \cdot m_\alpha = x$ and $r = x' + 1 \cdot m_\alpha$. Now, for a given m_α, if the challenger's response to the signing query at period t^* has $\sigma_2 \in \mathsf{good}_{m_\alpha}$, then the adversary cannot distinguish whether $\beta = 0$ or 1 information theoretically. In this case, the best the adversary can do (to cause the challenger to abort) is guess $\beta' \in \{0, 1\}$, compute $d' = \sigma_2 - \beta' \cdot m_\alpha$ and then output a signature with $\sigma_2^* = d + \beta' \cdot m_\alpha^*$. If its guess of this bit was correct, then this will force the challenger to abort. This occurs with 1/2 probability in this case. In the other case, if the challenger issues a signature for period t^* with $\sigma_2 \notin \mathsf{good}_{m_\alpha}$, then we also assume the challenger must abort. However, this corresponds to the event that σ_2 falls in the range $[2^\lambda - f(\lambda) + 1, 2^\lambda]$, which happens with at most negligible probability due to the challenger's selection of $d \in [0, 2^\lambda - f(\lambda)]$ (which adds only $\beta \cdot m_\alpha$, which can be at most $f(\lambda)$.)
. □

Claim 7.5. *If F is a secure pseudorandom function and $\lambda \geq 4$, then*

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 5] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 4] - \mathsf{negl}(\lambda).$$

Claim 7.6. *If F is a secure pseudorandom function and $T \in \mathsf{poly}(\lambda)$, then*

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 6] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 5] - \mathsf{negl}(\lambda).$$

Claim 7.7.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 7] = \frac{\mathbf{Adv}_{\mathcal{A}}[\text{Game } 6]}{\lambda^3 + \lambda^2}.$$

Claim 7.8.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 8] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 7].$$

Claim 7.9.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 9] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 8].$$

The proofs of Claims 7.5-7.9 are for programming the hash H_K. They are the same as those of Claims 6.5-6.9 in Appendix B (removing any mention of forking). □

Claim 7.10.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 10] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 9].$$

Proof. The attacker's view in these games is identical. The only difference is how the $\mathtt{state}_1$ value is computed, but both computations result in the same output and this follows from the correctness of $\mathsf{SeqShift}$. □

Claim 7.11.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 11] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 10].$$

Proof. The only difference in these games is how the public and secret key are computed, which we argue admits the same distribution. In Game 10, the bases for $\mathsf{SeqSetup}$ (which generates sk_1) and the bases for the pk are of the form g^{u_i}, where g is a generator of QR_N and each u_i is chosen randomly from $[1, N]$. In Game 11, a random $w \in \mathrm{QR}_N$ is chosen by the challenger and three of these g^{u_i} bases have the values w^d, w^β, w^{-1}, respectively, multiplied into them. Since these are all quadratic residues and the u_i values are chosen independently, the resulting distribution of this key pair will be the same in both games. □

Claim 7.12.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game } 12] = \mathbf{Adv}_{\mathcal{A}}[\text{Game } 11].$$

Proof. We argue in two cases that an attacker's advantage in these games is identical. First, in Game 12, when the break-in period $b \leq t^*$, then the challenger aborts and the attacker automatically loses. However, the same thing happens in Game 11. Due to Game 2, we know that the attacker only wins if the forgery period is t^*. Due to Game 1 (forward security game), we know that the attacker only wins if she forges for a period prior to the break-in. Thus, in this case, there is no way for the attacker to win, so aborting immediately and declaring the loss will not change anything.

Second, in Game 12, when $b > t^*$, the only change is how the secret key $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$ that the challenger must respond with is computed. In both games, the challenger computes $v_0 = g^{u_0} \cdot w^d$, $v_\alpha = g^{u_\alpha} \cdot w^\beta$, $\tilde{v} = g^{\tilde{u}} \cdot w^{-1}$ and for $i = 1$ to k, where $i \neq \alpha$, $v_i = g^{u_i}$. It returns $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$.

In Game 12, $\mathtt{state}_b$ is computed as $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_0^{\gamma}, \dots, v_k^{\gamma}, \tilde{v}^{\gamma}), b)$, where $\gamma = \prod_{i\in[1,b-1]} e_i$. Whereas in Game 11, $\mathtt{state}_1 = \mathsf{SeqSetup}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_0, \dots, v_k, \tilde{v}))$ and then $\mathtt{state}_{t+1} = \mathsf{SeqUpdate}(\mathtt{state}_t)$ for $t = 1$ to $b - 1$. By the correctness of $\mathsf{SeqProgram}$, these two methods of computing $\mathtt{state}_b$ will result in the same output. Thus, there is nothing for the adversary to distinguish in this case either. $\square$

Claim 7.13.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 13}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 12}].$$

Proof. The adversary's view in Games 12 and 13 is identical. The tuple $(s_0, \dots, s_k, \tilde{s})$ used to sign is the same in both games, the challenger just computes the same thing in two different ways. This follows from the correctness of $\mathsf{SeqSetup}$ and $\mathsf{SeqUpdate}$. $\square$

Claim 7.14.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 14}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 13}].$$

Proof. The adversary's view in Games 13 and 14 is identical. Per Assumption 3.1, $h = w^{e*} = w^{e_{t^*}} \bmod N$ for a random $w \in \mathrm{QR}_N$, so the distribution for the tuples $\mathsf{pk} = (U_0, \dots, U_k, \tilde{U})$ and $(s_0, \dots, s_k, \tilde{s})$ are identical. At the time of the break in, both games inherit from Game 12 that $b > t^*$ (otherwise, the challenger immediately aborts), so it is possible to compute those identical inputs to $\mathsf{SeqProgram}$ using h. $\square$

Claim 7.15.

$$\mathbf{Adv}_{\mathcal{A}}[\text{Game 15}] = \mathbf{Adv}_{\mathcal{A}}[\text{Game 14}].$$

Proof. The adversary's view in Games 14 and 15 is identical. The only change is how the signature query $(M = m_1|\dots|m_k, t^*)$ is computed, which we now show to be the same in both games. In both games (per Game 3), the challenger sets $\sigma_2 = d+\beta\cdot m_\alpha$. Let $E = \prod_{i\in[1,T]} e_i$ and $E' = \prod_{i\in[1,T]\setminus\{t^*\}} e_i$. In both games, $U_0 = g^{Eu_0} \cdot h^{E'd}$, $U_\alpha = g^{Eu_\alpha} \cdot h^{E'\beta}$, $\tilde{U} = g^{E\tilde{u}} \cdot h^{-E'}$ and for $i = 1$ to k, where $i \neq \alpha$, $U_i = g^{Eu_i}$. It then sets the public key information $\mathsf{pk} = (U_0, \dots, U_k, \tilde{U})$.

In Game 14, σ_1 is computed according to the normal signing algorithm. And observe from the construction that for fixed $\mathsf{pp}, \mathsf{pk}, M, t^*$ and σ_2, there is a single σ_1 value that will pass the verification equation (since σ_1 is the unique e^*th root modulo N of a deterministic function of these other values). Thus, it remains to show that the σ_1 of Game 15 will also pass the verification algorithm. In Game 15, σ_1 is computed as follows. Recall that $\sigma_2 = r = d + \beta \cdot m_\alpha$.

$$\begin{aligned}
\sigma_1 &= (g^{E'u_0}) \cdot (g^{E'\tilde{u}})^r \cdot \prod_{i=1}^{k} (g^{E'u_i m_i}) \\
&= (h^{E'/e^*})^{d+\beta m_\alpha - r} \cdot (g^{E'u_0}) \cdot (g^{E'\tilde{u}})^r \cdot \prod_{i=1}^{k} (g^{E'u_i m_i}) \\
&= (g^{E'u_0} \cdot h^{dE'/e^*}) \cdot (g^{E'\tilde{u}} \cdot h^{-E'/e^*})^r \cdot (g^{E'u_\alpha m_\alpha} \cdot h^{\beta m_\alpha E'/e^*}) \cdot \prod_{i=1, i\neq\alpha}^{k} (g^{E'u_i m_i})
\end{aligned}$$

So we have that:

$$
\begin{aligned}
\sigma_1^{e^*} &= (g^{E'u_0} \cdot h^{dE'/e^*})^{e^*} \cdot (g^{E'\tilde{u}} \cdot h^{-E'/e^*})^{re^*} \cdot (g^{E'u_\alpha m_\alpha} \cdot h^{\beta m_\alpha E'/e^*})^{e^*} \cdot \prod_{j=1, j\neq\alpha}^{k} (g^{E'u_j m_j})^{e^*} \\
&= (g^{Eu_0} \cdot h^{E'd}) \cdot (g^{E\tilde{u}} \cdot h^{-E'})^r \cdot (g^{Eu_\alpha} \cdot h^{E'\beta})^{m_\alpha} \cdot \prod_{j=1, j\neq\alpha}^{k} (g^{Eu_j})^{m_j} \\
&= U_0 \cdot \tilde{U}^r \cdot U_\alpha^{m_\alpha} \cdot \prod_{j=1, j\neq\alpha}^{k} U_j^{m_j} \\
&= U_0 \cdot \tilde{U}^{\sigma_2} \cdot \prod_{j=1}^{k} U_j^{m_j}
\end{aligned}
$$

Thus, in Game 15, this method for computing σ_1 results in the same value as Game 14. $\square$

7.4.1 Main Reduction

We now show that if there exists a polynomial-time (in λ) attacker that has advantage $\epsilon = \epsilon(\lambda)$ in Game 15, then there exists a polynomial-time (in λ) attacker for the RSA problem in Assumption 3.1 with advantage $\epsilon - \mathsf{negl}(\lambda)$.

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

Setup.

1. Obtain 1^T from the signature adversary $\mathcal{A}$.
2. Make random guesses of $t^* \in [1, T], \alpha \in [1, k], i^* \in [1, \lambda^3 + \lambda^2]$.
3. Make random choices of $d \in [0, 2^\lambda - f(\lambda)]$ and $\beta \in \{0, 1\}$.
4. Following [27] to program H_K, 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 $K = (K', c, e_{\text{default}})$. By construction, when i^* is the resolving index for t^*,

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

5. Set $\texttt{len} = k+2$. Choose a random $g \in \text{QR}_N$ and obtain $\texttt{state}_{\textsf{pp}} = \textsf{SeqSetup}(N, 1^T, K, 1^{\texttt{len}=k+2}, (g, g, \ldots, g))$.
6. Let $E = \prod_{i\in[1,T]} e_i$ and $E' = \prod_{i\in[1,T]\setminus\{t^*\}} e_i$. Compute $Y = g^E \mod N$.
7. Select random $u_0, \ldots, u_k, \tilde{u} \in [1, N]$.
8. Compute $U_0 = g^{Eu_0} \cdot h^{E'd}$, $U_\alpha = g^{Eu_\alpha} \cdot h^{E'\beta}$, $\tilde{U} = g^{E\tilde{u}} \cdot h^{-E'}$ and for $i = 1$ to k, where $i \neq \alpha$, $U_i = g^{Eu_i}$. It then sets the public key information $\textsf{pk} = (U_0, \ldots, U_k, \tilde{U})$.
9. Set the $\textsf{pp} = (T, N, Y, K, \texttt{state}_{\textsf{pp}})$. Send $(\textsf{pp}, \textsf{pk})$ to $\mathcal{A}$.

Queries. The adversary can make two different types of queries.

1. Signing: On input $(M = m_1|\ldots|m_k, t)$ from the adversary, if $t \neq t^*$, let $\gamma' = \prod_{i\in[1,T]\setminus\{t,t^*\}} e_i$ and $\gamma = \gamma' \cdot e^*$. The challenger computes $s_0 = g^{\gamma u_0} \cdot h^{\gamma' d}$, $s_\alpha = g^{\gamma u_\alpha} \cdot h^{\gamma' \beta}$, $\tilde{s} = g^{\gamma\tilde{u}} \cdot h^{-\gamma'}$

and for $i = 1$ to k, where $i \neq \alpha$, $s_i = g^{\gamma u_i}$. The signature algorithm then proceeds according to the regular algorithm by choosing a random $r \in [0, 2^\lambda - f(\lambda)]$, setting $\sigma_2 = r$ and $\sigma_1 = s_0 \cdot \tilde{s}^r \cdot \prod_{j=1}^{k} s_j^{m_j}$.

If $t = t^*$, the challenger sets $r = d + \beta \cdot m_\alpha$ (per Game 3) and then sets $\sigma_2 = r$ and computes

$$\sigma_1 \quad = \quad (g^{E' u_0}) \cdot (g^{E' \tilde{u}})^r \cdot \prod_{i=1}^{k} (g^{E' u_i m_i}).$$

The challenger responds with $\sigma = (\sigma_1, \sigma_2)$.

2. Break-In: On input a break-in key request for period b from the adversary, if $b \leq t^*$, then the challenger aborts. If $b > t^*$, then the challenger computes $\pi' = \prod_{i \in [1, b-1] \setminus \{t^*\}} e_i$ and $\pi = \pi' \cdot e^*$ and then $v_0 = g^{\pi u_0} \cdot h^{\pi' d}$, $v_\alpha = g^{\pi u_\alpha} \cdot h^{\pi' \beta}$, $\tilde{v} = g^{\pi \tilde{u}} \cdot h^{-\pi'}$ and for $i = 1$ to k, where $i \neq \alpha$, $v_i = g^{\pi u_i}$. It returns $\mathsf{sk}_b = (\mathtt{state}_b, e_b, b)$, where $\mathtt{state}_b$ is computed as $\mathtt{state}_b = \mathsf{SeqProgram}(N, 1^T, K, 1^{\mathtt{len}=k+2}, (v_0, \ldots, v_k, \tilde{v}), b)$.

Output. Eventually, the adversary outputs a tuple $(M^* = m_1^* | \ldots | m_k^*, t, \sigma^* = (\sigma_1^*, \sigma_2^*))$. If the signature tuple does not verify, if any of the challenger's guesses of t^*, α, i^* were incorrect, if t is at or after the break-in b, the message-period pair of (M^*, t) was previously queried to the signing oracle or $\sigma_2^* = d + \beta m_\alpha^*$, then abort. Let $M = m_1 | \ldots | m_k$ and $\sigma = (\sigma_1, \sigma_2)$ be the message and signature associated with the adversary's signing query on period t^*. If $\sigma_2 \notin \mathsf{good}_{m_\alpha}$, then abort. These abort conditions are all consistent with the adversary not winning Game 15.

Otherwise, we have an equation of the form:

$$\begin{aligned}
(\sigma_1^*)^{e^*} &= U_0 \cdot \tilde{U}^{\sigma_2^*} \cdot \prod_{j=1}^{k} U_j^{m_j^*} = U_0 \cdot \tilde{U}^{\sigma_2^*} \cdot U_\alpha^{m_\alpha^*} \cdot \prod_{j=1, j\neq\alpha}^{k} U_j^{m_j^*} \\
&= (g^{E u_0} \cdot h^{E' d}) \cdot (g^{E \tilde{u}} \cdot h^{-E'})^{\sigma_2^*} \cdot (g^{E u_\alpha} \cdot h^{E' \beta})^{m_\alpha^*} \prod_{j=1, j\neq\alpha}^{k} (g^{E u_i})^{m_j^*} \\
&= h^{E'(d + \beta m_\alpha^* - \sigma_2^*)} \cdot g^{E \cdot (u_0 + \tilde{u} \sigma_2^* + \sum_{i=1}^{k} u_i m_i^*)} \\
&= h^{E'(d + \beta m_\alpha^* - \sigma_2^*)} \cdot g^{e^* \cdot E' \cdot (u_0 + \tilde{u} \sigma_2^* + \sum_{i=1}^{k} u_i m_i^*)}
\end{aligned}$$

Thus, we can divide both sides by the same amount to obtain the equation:

$$\left(\sigma_1^* \cdot g^{-E' \cdot (u_0 + \tilde{u} \sigma_2^* + \sum_{i=1}^{k} u_i m_i^*)}\right)^{e^*} = h^{E'(d + \beta m_\alpha^* - \sigma_2^*)}$$

View this equation as $x^{e^*} = h^{b'}$ for $x = \sigma_1^* \cdot g^{-E' \cdot (u_0 + \tilde{u} \sigma_2^* + \sum_{i=1}^{k} u_i m_i^*)}$ and $b' = E'(d + \beta m_\alpha^* - \sigma_2^*)$. We know from Game 4 that $b' \neq 0$. Assume b' is non-negative (if not, raise both sides of the equation to -1.) Now, the challenger can apply Lemma 3.2 to the above to efficiently obtain a $z \in \mathbb{Z}_N$ such that $z^{e^*} = h$. The challenger outputs z as the solution to the RSA challenge.

Analysis. The attacker's view in the main reduction is the same as Game 15. It remains to argue that $\gcd(e^*, b') = 1$, so that the challenger will be able to apply Lemma 3.2 in the final step of the main reduction. From Games 5 and 6, we have that e^* is relatively prime to E'. We have that

$b' = d + \beta m_\alpha^* - \sigma_2^* > 1$. Since $d, \sigma_2^* \in [0, 2^\lambda - f(\lambda)]$, $\beta \in \{0,1\}$ and $m_\alpha^* \leq f(\lambda)$, we know that the value $d + \beta m_\alpha^* - \sigma_2^*$ cannot exceed 2^λ. Given that e^* was chosen from the range $[2^\lambda, 2^{\lambda+1} - 1]$, there is no possibility that $e^* = d + \beta m_\alpha^* - \sigma_2^*$, since 2^λ is not prime. Therefore, we can conclude that $\gcd(e^*, b') = 1$ and the challenger will be able to compute the RSA solution.

8 Multiple Signatures per Update

We move to showing how to transform our scheme to allow for multiple signatures per time period. We use an idea, which is common in literature, to add a level of indirection by having the forward secure scheme sign a verification key as its message. Then the standard signature scheme can be used to sign as many signatures in the time period as needed, with the understanding that if the secret key becomes compromised by an attacker during period t, then all signatures from that period (and later) are suspect. As discussed before arguably the best forward secure key hygiene is to do a key update on every single sign which enables forward security at the finest granularity possible on key compromise. However, we include this transformation to give options when deploying a system. The cost of this transformation is an increase in signature size. The exact increase will depend on the public key size of the underlying signature scheme and for our regime at least double it.

The transformation is described below.

Given a weakly secure, single sign forward secure signature scheme $\Pi_{\mathsf{weak}} = (\mathsf{Setup}_{\mathsf{weak}}, \mathsf{KeyGen}_{\mathsf{weak}}, \mathsf{Update}_{\mathsf{weak}}, \mathsf{Sign}_{\mathsf{weak}}, \mathsf{Verify}_{\mathsf{weak}})$ and a regular signature scheme scheme $\Pi_{\mathsf{reg}} = (\mathsf{KeyGen}_{\mathsf{reg}}, \mathsf{Sign}_{\mathsf{reg}}, \mathsf{Verify}_{\mathsf{reg}})$ that is existentially unforgeable with respect to adaptive chosen message attacks [20], there is a generic method for obtaining an adaptively secure, multi-sign forward secure signature scheme $\Pi_{\mathsf{full}} = \mathsf{Setup}_{\mathsf{full}}, \mathsf{KeyGen}_{\mathsf{full}}, \mathsf{Update}_{\mathsf{full}}, \mathsf{Sign}_{\mathsf{full}}, \mathsf{Verify}_{\mathsf{full}})$. This method works as follows. Let the message space for Π_{weak} be the public key space of Π_{reg}. Let the message spaces of Π_{full} and Π_{reg} be the same. In practice, one could always first apply a collision-resistant hash function to map inputs to the proper message space.

$\mathsf{Setup}_{\mathsf{full}}(1^\lambda, 1^T)$ Run $\mathsf{Setup}_{\mathsf{weak}}(1^\lambda, 1^T)$ to obtain pp.

$\mathsf{KeyGen}_{\mathsf{full}}(\mathsf{pp})$ On input pp, run $\mathsf{KeyGen}_{\mathsf{weak}}(\mathsf{pp})$ to obtain $(\mathsf{pk}, \mathsf{sk}'_1)$ and $\mathsf{KeyGen}_{\mathsf{reg}}(1^\lambda)$ to obtain $(\mathsf{pk}', \mathsf{sk}')$. Compute $\sigma' = \mathsf{Sign}_{\mathsf{weak}}(\mathsf{pp}, \mathsf{sk}'_1, \mathsf{pk}')$. The public key is pk and the secret key is $\mathsf{sk}_1 = (\mathsf{sk}'_1, \mathsf{pk}', \mathsf{sk}', \sigma')$.

$\mathsf{Update}_{\mathsf{full}}(\mathsf{pp}, \mathsf{sk}_t)$ Parse $\mathsf{sk}_t = (\mathsf{sk}'_t, \mathsf{pk}', \mathsf{sk}', \sigma')$. Run $\mathsf{Update}_{\mathsf{weak}}(\mathsf{pp}, \mathsf{sk}'_t)$ to obtain sk'_{t+1}. Run $\mathsf{KeyGen}_{\mathsf{reg}}(1^\lambda)$ to obtain $(\mathsf{pk}'', \mathsf{sk}'')$. Compute $\sigma'' = \mathsf{Sign}_{\mathsf{weak}}(\mathsf{pp}, \mathsf{sk}'_{t+1}, \mathsf{pk}'')$. Output $\mathsf{sk}_{t+1} = (\mathsf{sk}'_{t+1}, \mathsf{pk}'', \mathsf{sk}'', \sigma'')$.

$\mathsf{Sign}_{\mathsf{full}}(\mathsf{pp}, \mathsf{sk}_t, m)$ Parse $\mathsf{sk}_t = (\mathsf{sk}'_t, \mathsf{pk}', \mathsf{sk}', \sigma')$. Compute $s = \mathsf{Sign}_{\mathsf{reg}}(\mathsf{sk}', m)$. Output the signature $\sigma = (\mathsf{pk}', \sigma', s)$.

$\mathsf{Verify}_{\mathsf{full}}(\mathsf{pp}, \mathsf{pk}, m, t, \sigma = (\mathsf{pk}', \sigma', s))$ Accept iff $\mathsf{Verify}_{\mathsf{reg}}(\mathsf{pk}', m, s) = \mathsf{Verify}_{\mathsf{weak}}(\mathsf{pp}, \mathsf{pk}, \mathsf{pk}', t, \sigma') = 1$.

Theorem 8.1. *If* Π_{weak} *is a weakly secure single sign forward secure signature scheme and* Π_{reg} *is a regular signature scheme that is existentially unforgeable with respect to adaptive chosen message attacks, then the above key-evolving signature construction* Π_{full} *is forward secure.*

Proof of the above theorem is rather straightforward and we omit it here. Roughly, any adversary that can win in the forward security game with Π_{full} with non negligible probability can be used to break either the weakly secure single sign forward security of Π_{weak} or the regular security of Π_{reg} with non negligible probability. We can split such an attacker into new types. In the first type the attacker with some non-negligible probability successfully forges on period t^* by producing a valid forward secure certificate/signature that was different than the one issued by the legitimate signer. In this case the attacker can be utilized as an attacker on the weak forward secure signature scheme. Here the weakness is not problem since the standard public key for the legitimate temporary signature at t^* can be chosen by the reduction algorithm in advance of receiving the challenge key for the forward secure scheme. Next, if the attacker is not of the type above then when it is successful with all but negligible probability it forges using the same temporary public key at t^*. This can be readily transformed into an attack on the standard signature scheme.

Instantiating the Scheme In choosing the standard scheme one would like to remain consistent and choose a hash-and-sign scheme that is provably secure from an RSA-based assumption in the standard model. There exist multiple choices (e.g [25, 26, 23]). One obvious optimization is that the weakly secure scheme should sign the output of the collision resistant hash of the temporary verification key. Still the new signature will have to include the description of the temporary verification key which will include a description of a new (temporary) RSA modulus N' as well as a few group elements of $\mathbb{Z}_{N'}^*$, and other keying material depending on the scheme. Cumulatively, these could significantly increase the signature size in practice. One mitigating strategy is to push as much of that public key material into some additional common parameters. Ideally, one could push almost all material except the new temporary modulus N'. For instance, group elements could be derived from large integers from setup and moding them by N'. The details of such modifications will vary on a scheme by scheme basis and are outside the scope of this paper.

9 Performance Evaluation

We now analyze the performance of the two main forward-secure schemes presented. In particular, we look at the random oracle based construction from Section 6 and the standard model construction from Section 7. The latter construction has the single sign restriction, meaning that the Update algorithm must be run after each signature generated; however, as our numbers will support key update operations will be cheap enough to still support a high rate of signing. And if more frequent signing is required one could move to the hybrid model of Section 8. We provide this analysis to give the reader some intuitive idea of the concrete costs of our cryptosystems, however, we emphasize that this does not represent an exhaustive search of reasonable configurations of our system.

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 [41]. 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

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

Figure 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.

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	$1 \cdot \mathbb{P}_{\lvert e\rvert} + 1 \cdot \mathbb{E}_{\lvert e\rvert} + 1 \cdot \mathbb{E}_{\lvert\sigma_2\rvert} + 1 \cdot \mathbb{M}$	0.73ms	0.73ms	0.73ms	0.73ms	0.73ms	0.73ms

Figure 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.

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.

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.[10] These evaluations will be considered for a maximum number of periods of

[10]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.

Sec.7	Operation	Parameters			Time when $T =$					
Alg.	Count	k	$\|e\|$	$\|\sigma_2\|$	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	$1 \cdot \mathbb{P}_{\|e\|} + (k+2) \cdot$	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
	$(k+2) \lg T \cdot \mathbb{E}_{\|e\|}$	8	113	112	41.01ms	54.67ms	68.34ms	82.01ms	95.68ms	0.11s
		256	82	81	0.70s	0.94s	1.17s	1.41s	1.64s	1.87s
Sign	$k \cdot \mathbb{E}_{\ell} + 1 \cdot \mathbb{E}_{\|\sigma_2\|}+$	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	$1 \cdot \mathbb{P}_{\|e\|} + k \cdot \mathbb{E}_{\ell}+$	1	337	336	4.02ms	4.02ms	4.02ms	4.02ms	4.02ms	4.02ms
	$1 \cdot \mathbb{E}_{\|\sigma_2\|} + 1 \cdot \mathbb{E}_{\|e\|}$	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

Figure 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.

$T \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}, 2^{32}\}$.[11] The Setup algorithm computes the modular multiplications with respect to $\phi(N)$ while the other algorithms due 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.

9.1 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 [12], 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

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

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

Sec. 6		Space when $T =$					
Item	Element Count	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	$(2\lg T)\mathbb{Z}_N + 1\lvert e\rvert$	6.0K	8.0K	10.0K	12.0K	14.0K	16.0K
σ	$1\mathbb{Z}_N + 1\lvert\sigma_2\rvert$	0.26K	0.26K	0.26K	0.26K	0.26K	0.26K

Figure 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 `len` $= k+2$ generators are the same, so we use the optimization from Section 5.1.

constructions, the answer is favorable. First, there are efficient algorithms for generating RSA moduli that distribute trust across multiple parties [12], 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 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 of Section 8 where the tradeoff will be 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.

Sec.7	Element	Parameters			Space when $T =$					
Item	Count	k	$\|e\|$	$\|\sigma_2\|$	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

Figure 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 `len` $= k+2$ generators are the same, so we use the optimization from Section 5.1.

9.2 A Quick Look at an MMM Variant

We conclude with a brief examination and comparison to a variant of the MMM [34] forward secure scheme which also supports standard model instantiations. For such a purpose, it is important to consider MMM in a fair context for comparison: (1) While a nice feature of MMM is that it allows for forward secure signatures with no apriori bound on T, for comparison it is useful to consider a slimmed down version of MMM that knows such a bound ahead of time. To that end, we consider using MMM with just a single tree of size T built from the "sum" composition. (2) Since our scheme allows for a global setup algorithm, we should consider if such a setup could help improve the signature sizes in the MMM variant. (3) The base signature scheme in the variant should be provably secure from the RSA assumption in the standard model.

With this in mind we consider an MMM variant which is built from a single tree of depth $\lg(T)$ and has T leaves. The public key for the base signature scheme will be an RSA modulus and use the a similar signature structure as we described in Section 7. While this scheme requires an additional $k+1$ group elements, we assume that these can be derived from a common reference string. We briefly examine the signature size and key generation time of such a scheme.

First, any signature in their scheme will consist of 2 public keys, 2 signatures and $\lg T$ hash values. If public keys be are RSA moduli of 2048 bits and the hash is 256 bit SHA, then signatures would be approximately of size $2 \times 2048 + 2(2048 + 80) + \lg(T) \times 256$ bits. If we aim for high frequency signing/updates and set $T = 2^{32}$, then we get 16544 bits or 2068 bytes. This is almost a factor of 10 more than the approximately 260 bytes we report in Figure 5. Even if we go to the other end of the spectrum and plug in $T = 2^{12}$, the above version of [34] gives us signatures of 1428 bytes which is a factor of 5.5 more than 260.

For key generation, the MMM scheme must build a Merkle tree which requires generating the public keys for all leaves in the system. This will be dominated by the cost of two 1024 bit prime searches for each leaf or $2 \times T$ 1024 bit prime searches total. For a value of $T = 2^{32}$, this scheme becomes impractical with a key generation time of roughly 7.8 years (see Figure 1). Compare this with our scheme (see Figure 3), where after a one-time global setup that takes days (but not years), each key generation takes only seconds.

We remark that there could be other options for the core signature scheme that tradeoff signature size versus key generation time. In addition, we note that the Bellare-Miner [8] tree based scheme would save significantly on key generation relative to the MMM variant since it does not need to start by building a Merkle tree, but would require $\lg(T)$ RSA-based signatures in each signature (versus $\lg(T)$ hash outputs).

Thus, we conclude that our construction offers significant efficiency savings over these prior RSA standard model constructions.

References

[1] Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. From identification to signatures via the Fiat-Shamir transform: Necessary and sufficient conditions for security and forward-security. *IEEE Trans. Information Theory*, 54(8):3631–3646, 2008.

[2] Michel Abdalla, Fabrice Ben Hamouda, and David Pointcheval. Tighter reductions for forward-secure signature schemes. In *Public-Key Cryptography - PKC*, pages 292–311, 2013. Full version available at `https://eprint.iacr.org/2017/746`.

[3] Michel Abdalla, Fabrice Benhamouda, and David Pointcheval. On the tightness of forward-secure signature reductions. *J. Cryptology*, 32(1):84–150, 2019.

[4] Michel Abdalla, Sara K. Miner, and Chanathip Namprempre. Forward-secure threshold signature schemes. In *CT-RSA*, pages 441–456, 2001.

[5] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In *Advances in Cryptology - ASIACRYPT*, pages 116–129, 2000.

[6] Ross Anderson. Invited Lecture. Fourth Annual Conference on Computer and Communications Security, ACM, 1997.

[7] Giuseppe Ateniese and Breno de Medeiros. Identity-based chameleon hash and applications. In *Financial Cryptography '04*, volume 3110, pages 164–180, 2004.

[8] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *Advances in Cryptology - CRYPTO*, pages 431–448, 1999.

[9] Mihir Bellare and Todor Ristov. Hash functions from sigma protocols and improvements to VSH. In *ASIACRYPT*, pages 125–142, 2008.

[10] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT*, pages 274–285, 1993.

[11] Dan Boneh and Xavier Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In *EUROCRYPT*, volume 3027, pages 223–238, 2004.

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

[13] Xavier Boyen, Hovav Shacham, Emily Shen, and Brent Waters. Forward-secure signatures with untrusted update. In *ACM CCS*, pages 191–200, 2006.

[14] Jan Camenisch and Maciej Koprowski. Fine-grained forward-secure signature schemes without random oracles. *Discrete Applied Mathematics*, 154(2):175–188, 2006.

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

[16] Eric Cronin, Sugih Jamin, Tal Malkin, and Patrick D. McDaniel. On the performance, feasibility, and use of forward-secure signatures. In *ACM CCS*, pages 131–144, 2003.

[17] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[18] Marc Fischlin. The Cramer-Shoup strong-RSA signature scheme revisited. In *Public Key Cryptography - PKC 2003*, pages 116–129, 2003.

[19] 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/`.

[20] 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.

[21] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" identity-based signature scheme resulting from zero-knowledge. In *Advances in Cryptology - CRYPTO*, pages 216–231, 1988.

[22] 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`.

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

[24] Dennis Hofheinz and Eike Kiltz. Programmable hash functions and their applications. In *Advances in Cryptology - CRYPTO 2008*, pages 21–38, 2008.

[25] Susan Hohenberger and Brent Waters. Realizing hash-and-sign signatures under standard assumptions. In *EUROCRYPT '09*, volume 5479 of LNCS, pages 333–350, 2009.

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

[27] Susan Hohenberger and Brent Waters. Synchronized aggregate signatures from the RSA assumption. In *EUROCRYPT*, volume 10821, pages 197–229, 2018.

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

[29] Anton Kozlov and Leonid Reyzin. Forward-secure signatures with fast key update. In *Security in Communication Networks*, pages 241–256, 2002.

[30] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *ACM Conference on Computer and Comm. Security*, pages 108–115, 2000.

[31] Hugo Krawczyk and Tal Rabin. Chameleon signatures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2000.

[32] Benoît Libert, Jean-Jacques Quisquater, and Moti Yung. Forward-secure signatures in untrusted update environments: efficient and generic constructions. In *ACM CCS*, pages 266–275, 2007.

[33] Benoît Libert and Moti Yung. Dynamic fully forward-secure group signatures. In *ASIACCS*, pages 70–81, 2010.

[34] 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.

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

[36] Payman Mohassel. One-time signatures and chameleon hash functions. In *Selected Areas in Cryptography - 17th International Workshop, SAC*, pages 302–319, 2010.

[37] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In *Advances in Cryptology - EUROCRYPT*, pages 387–398, 1996.

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

[39] 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.

[40] Adi Shamir. On the generation of cryptographically strong pseudorandom sequences. *ACM Transaction on Computer Systems*, 1:38–44, 1983.

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

[42] Dawn Xiaodong Song. Practical forward secure group signature schemes. In *ACM Conference on Computer and Communications Security*, pages 225–234, 2001.

A Proof of Lemma 5.1

We here prove Lemma 5.1 which was the anchor in proving correctness of our RSA sequencer of Section 5. The proof of this lemma follows very closely to the proof of correctness of the log storage scheme from [27] and we mainly need to update it to account for storing tuples of group elements for $\texttt{len} > 1$.

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

Claim A.1. *Suppose that our state description, as stated in the statement of Lemma 5.1, is accurate for period* $\texttt{index}$*. Consider an update operation where the period moves from* $\texttt{index}$ *to* $\texttt{index} + 1$*. This will result in a 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 $\texttt{index}$ mod 2^i. It will trigger a pass down operation only when $\texttt{count}$ increments to $\texttt{count} = 2^{i-1}$. Similarly, in Case 3 there is a tuple with $\texttt{count} = r - 2^{i-1}$. A pass down operation is only triggered when it increments to 2^i which means $\texttt{index} + 1$ is a multiple of 2^{i-1}. □

Claim A.2. *Suppose that our state description, as stated in the statement of Lemma 5.1, is accurate for period* $\texttt{index}$ *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* $\texttt{index}$*. (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 $\texttt{index}$ where $T - \texttt{index} \leq 2^{i+1} - 2$. It then follows that $z + 2^i - 1$ is the smallest value of $\texttt{index}$ where $T - \texttt{index} \leq 2^{i+1} - 2$ *and* $\texttt{index}$ mod 2^i. Now let's consider the next value 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 $\texttt{index}$ if the state description was valid at $\texttt{index}$, 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 $\texttt{index}$ then in $\texttt{index} + 1$ it will match the state description.

Claim A.3. *Suppose at period* $\texttt{index}$ *the state description is accurate, as stated in the statement of Lemma 5.1, 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 $\texttt{index}$ then S_{i+1} must also be empty. Since S_{i+1} is empty, there is nothing to pass down. □

Claim A.4. *Suppose at period* $\texttt{index}$ *the state description is accurate, as stated in the statement of Lemma 5.1, 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 $\texttt{index}$ we are in Case 2 there are two tuples in S_i where the one with the smaller $\texttt{open}$ value is of the form $((w_1 = v_1^{\prod_{j \in T \setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j \in T \setminus R} e_j}), \texttt{open} = (k+1) \cdot 2^i - 1, \texttt{closing} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r)$. The update algorithm will increment $\texttt{count}$ to $r+1$ and update $w_1, \ldots, w_{\texttt{len}}$ to $w_1 = w_1^{e_{\texttt{closing}+\texttt{count}}}, \ldots, w_{\texttt{len}} = w_{\texttt{len}}^{e_{\texttt{closing}+\texttt{count}}}$ which gives the needed form to remain in Case 2. The second tuple is of the form $((w_1 = v_1^{\prod_{j \in T \setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j \in T \setminus R} e_j}), \texttt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{closing} = (k+1) \cdot 2^i - 1, \texttt{count} = 0)$. The update algorithm will not modify it as the other tuple had the smaller $\texttt{open}$ 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 A.1) that $\texttt{index} \mod 2^i = r \neq 2^i - 1$ and that a pushdown would only happen as $\texttt{index}$ transfers to being a multiple of 2^i.

We now consider the subcase where $r = 2^{i-1} - 1$ at $\texttt{index}$ 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 $((w_1 = v_1^{\prod_{j \in T \setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j \in T \setminus R} e_j}), \texttt{open} = (k+1) \cdot 2^i - 1, \texttt{closing} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{count} = r = 2^{i-1} - 1)$. The update operation will first modify the tuple to a new $\texttt{count}$ 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 A.1. □

Claim A.5. *Suppose at period* $\texttt{index}$ *the state description is accurate, as stated in the statement of Lemma 5.1, 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 $((w_1 = v_1^{\prod_{j \in T \setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j \in T \setminus R} e_j}), \texttt{open} = (k+1) \cdot 2^i - 1 + 2^{i-1}, \texttt{closing} = (k+1) \cdot 2^i - 1, \texttt{count} = r - 2^{i-1})$. The update algorithm will increment $\texttt{count}$ to $r+1$ and update $w_1, \ldots, w_{\texttt{len}}$ as $w_1 = w_1^{e_{\texttt{closing}+\texttt{count}}}, \ldots, w_{\texttt{len}} = w_{\texttt{len}}^{e_{\texttt{closing}+\texttt{count}}}$ 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 A.2. 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:

$$((w_1 = v_1^{\prod_{j\in T\setminus R} e_j}, \ldots, w_{\texttt{len}} = v_{\texttt{len}}^{\prod_{j\in T\setminus R} e_j}), \texttt{open} = (\tilde{k}+1)\cdot 2^{i+1}-1,$$
$$\texttt{closing} = (\tilde{k}+1)\cdot 2^{i+1} - 1 + 2^i, \texttt{count} = 2^i - 1).$$

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 $w_1, \ldots, w_{\texttt{len}}$ 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, $\texttt{index}+1 = 2^i\cdot k'$ where $k' = k+1$. To match the description for $\texttt{index}+1$ 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 at $\texttt{index}+1$ 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. □

B Proof of Hash Programming Claims 6.5, 6.6, 6.7, 6.8 and 6.9

The proof techniques that we use to program the (non-random oracle) hash function H_K in the proof of Theorem 6.1 for our forward-secure signatures come from those used by [27] in the aggregate signature context. We include the proofs of these claims from [27] here for completeness, making only minor modifications to address the fork in Game 4.

B.1 Proof of Claim 6.5

Proof. Recall that in Game 4 the attacker will only win if both forks have the same forgery period t^*. 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. □

B.2 Proof of Claim 6.6

Proof. Recall that both forks will share the same forgery period t^*. Games 5 and 6 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. □

B.3 Proof of Claim 6.7

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. □

B.4 Proof of Claim 6.8

Proof. In Game 7, c is chosen randomly in $\{0,1\}^\lambda$. In Game 8, 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 6. $\square$

B.5 Proof of Claim 6.9

Proof. An adversary's advantage in these games is the same. In Game 8, the attacker could only win if $2^\lambda + y'$ was a prime, and thus the distributions are the same. $\square$