Forward-Secure Signatures with Optimal Signing and Verifying

Gene Itkis[1] and Leonid Reyzin[2]

[1] Boston University Computer Science Dept.
111 Cummington St.
Boston, MA 02215, USA
itkis@bu.edu

[2] Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
reyzin@theory.lcs.mit.edu
http://theory.lcs.mit.edu/~reyzin

Abstract. We propose the first forward-secure signature scheme for which both signing and verifying are as efficient as for one of the most efficient ordinary signature schemes (Guillou-Quisquater [GQ88]), each requiring just two modular exponentiations with a short exponent. All previously proposed forward-secure signature schemes took significantly longer to sign and verify than ordinary signature schemes.

Our scheme requires only fractional increases to the sizes of keys and signatures, and no additional public storage. Like the underlying [GQ88] scheme, our scheme is provably secure in the random oracle model.

1 Introduction

THE PURPOSE OF FORWARD SECURITY. Ordinary digital signatures have a fundamental limitation: if the secret key of a signer is compromised, all the signatures (past and future) of that signer become worthless. This limitation undermines, in particular, the non-repudiation property that digital signatures are often intended to provide. Indeed, one of the easiest ways for Alice to repudiate her signatures is to post her secret key anonymously somewhere on the Internet and claim to be a victim of a computer break-in. In principle, various revocation techniques can be used to prevent users from accepting signatures with compromised keys. However, even with these techniques in place, the users who had accepted signatures *before* the keys were compromised are now left at the mercy of the signer, who could (and, if honest, would) re-issue the signatures with new keys.

Forward-secure signature schemes, first proposed by Anderson in [And97] and formalized by Bellare and Miner in [BM99], are intended to address this limitation. Namely, the goal of a forward-secure signature scheme is to preserve the validity of *past* signatures even if the current secret key has been compromised. This is accomplished by dividing the total time that given public key is valid into T *time periods*, and using a different secret key in each time period (while the public key remains fixed). Each subsequent secret key is computed from the current secret key via a *key update* algorithm. The time period during which a message is signed becomes part of the signature. Forward security property means that even if the current secret key is compromised, a forger cannot forge signatures for past time periods.

Prior Schemes. Prior forward-secure signature schemes can be divided into two categories: those that use arbitrary signature schemes in a black-box manner, and those that modify specific signature scheme.

In the first category, the schemes use some method in which a master public key is used to certify (perhaps via a chain of certificates) the current public key for a particular time period. Usually, these schemes require increases in storage space by noticeable factors in order to maintain the current (public) certificates and the (secret) keys for issuing future certificates. They also require longer verification times than ordinary signatures do, because the verifier needs to verify the entire certificate chain in addition to verifying the actual signature on the message. There is, in fact, a trade-off between storage space and verification time. The two best such schemes are the tree-based scheme of Bellare and Miner [BM99][1] (requiring storage of about $\log_2 T$ secret keys and non-secret certificates, and verification of about $\log_2 T$ ordinary signatures) and the scheme of Krawczyk [Kra00] (requiring storage of T non-secret certificates, and verification of only 2 ordinary signatures).

In the second category, there have been two schemes proposed so far (both in the random oracle model): the scheme of Bellare and Miner [BM99] based on the Fiat-Shamir scheme [FS86], and the scheme of Abdalla and Reyzin [AR00] based the 2^t-th root scheme [OO88,OS90,Mic94]. While needing less space than the schemes in the first category, both [BM99] and [AR00] require signing and verification times that are linear in T.

Our Results. We propose a scheme in the second category, based on one of the most efficient ordinary signature schemes, due to Guillou-Quisquater [GQ88]. It uses just two modular exponentiations with short exponents for both signing and verifying.

Ours is the first forward-secure scheme where *both signing and verifying are as efficient as the underlying ordinary signature scheme*. Moreover, in our scheme *the space requirements for keys and signatures are nearly the same as those in the underlying signature scheme* (for realistic parameter values, less than 50% more).

The price of such efficient signing and verifying and storage is in the running times of our key generation and update routines: both are linear in T (however, so is the key generation and non-secret storage in the scheme of [Kra00]; as well as the key generation, signing and verifying in the Fiat-Shamir-based scheme of [BM99] and the scheme of [AR00]). However, key generation and update are

[1] Some improvements to tree-based scheme of [BM99] (not affecting this discussion) have been proposed in [AR00] and [MI].

(presumably) performed much less frequently than signing and verifying, and can be performed off-line as long in advance as necessary. Moreover, we show that, if we are willing to tolerate secret storage of $1 + \log_2 T$ values, we can reduce the running time of the key update algorithm to be logarithmic in T without affecting the other components (this, rather unexpectedly, involves an interesting application of pebbling). For realistic parameter values, the total storage requirements, even with these additional secrets, are still less than in all prior schemes; the only exception is the [AR00] scheme, which has very inefficient signing and verifying.

Our scheme is provably secure in the random oracle model based on a variant of the strong RSA assumption (precisely defined in Section 2.2).

2 Background

2.1 Definitions

This section closely follows the first formal definition of forward-secure signatures proposed by Bellare and Miner [BM99]. Their definition, in turn, is based on the Goldwasser, Micali and Rivest's [GMR88] definition of (ordinary) digital signatures secure against adaptive chosen message attacks.

KEY EVOLUTION. The approach taken by forward-secure schemes is to change the secret key periodically (and require the owner to properly destroy the old secret key[2]). Thus we consider time to be divided into time periods; at the end of each time period, a new secret key is produced and the old one is destroyed. The number of the time period when a signature was generated is part of the signature and is input to the verification algorithm; signatures with incorrect time periods should not verify.

Of course, while modifying the secret key, one would like to keep the public key fixed. This can, for example, be achieved by use of a "master" public key, which is somehow used to certify a temporary public key for the current time period (note however, than one needs to be careful not to keep around the corresponding "master" secret key—its presence would defeat the purpose of forward security) . The first simple incarnation of this approach was proposed by [And97]; a very elegant tree-based solution was proposed by [BM99]; another approach, based on generating all of the certificates in advance, was put forward by [Kra00]. However, in general, one can conceive of schemes where the public

[2] Obviously, if the key owner does not properly destroy her old keys, an attacker can obtain them and thus forge the "old" signatures. Moreover, if the key owner does not detect that the current key was leaked, the attacker may hold on to the compromised key for a few time periods, and forge "old" signatures then. Indeed, proper deletion of the old keys and proper intrusion detection are non-trivial tasks. However, it is reasonable to insist that the key owner perform such deletion and intrusion detection—certainly more reasonable than insisting that she guarantee the secrecy of her active keys through resistance to any intrusion attack.

key stays fixed but no such certificates of per-period public keys are present (and, indeed, such schemes are proposed in [BM99,AR00], as well as in this paper).

The notion of a *key-evolving* signature scheme captures, in full generality, the idea of a scheme with a fixed public key and a varying secret key. It is, essentially, a regular signature scheme with the additions of time periods and the key update algorithm. Note that this notion is purely functional: security is addressed separately, in the definition of forward security (which is the appropriate security notion for key-evolving signature schemes).

Thus, a *key-evolving digital signature scheme* is a quadruple of algorithms, $\mathsf{FSIG} = (\mathsf{FSIG.key}, \mathsf{FSIG.sign}, \mathsf{FSIG.ver}, \mathsf{FSIG.update})$, where:

- $\mathsf{FSIG.key}$, the *key generation* algorithm, is a probabilistic algorithm which takes as input a security parameter $k \in \mathsf{N}$ (given in unary as 1^k) and the total number of periods T and returns a pair (SK_1, PK), the initial secret key and the public key;
- $\mathsf{FSIG.sign}$, the (possibly probabilistic) *signing* algorithm, takes as input the secret key $SK_j = \langle S_j, j, T\rangle$ for the time period $j \leq T$ and the message M to be signed and returns the signature $\langle j, sign\rangle$ of M for time period j;
- $\mathsf{FSIG.ver}$, the (deterministic) *verification* algorithm, takes as input the public key PK, a message M, and a candidate signature $\langle j, sign\rangle$, and returns 1 if $\langle j, sign\rangle$ is a *valid* signature of M or 0, otherwise. It is required that $\mathsf{FSIG.ver}(PK, M, \mathsf{FSIG.sign}(SK_j, M)) = 1$ for every message M and time period j.
- $\mathsf{FSIG.update}$, the (possibly probabilistic) *secret key update* algorithm, takes as input the secret key SK_j for the current period $j < T$ and returns the new secret key SK_{j+1} for the next period $j + 1$.

We adopt the convention that SK_{T+1} is the empty string and $\mathsf{FSIG.update}(SK_T)$ returns SK_{T+1}.

When we work in the random oracle model, all the above-mentioned algorithms would have an additional security parameter, 1^l, and oracle access to a public hash function $H : \{0,1\}^* \rightarrow \{0,1\}^l$, which is assumed to be random in the security analysis.

Forward Security. Forward security captures the notion that it should be computationally infeasible for any adversary to forge a signature for any past time period even in the event of exposure of the current secret key. Of course, since the update algorithm is public, nothing can be done with respect to future secret keys, except for revoking the public key (thus invalidating all signatures for the time period of the break-in and thereafter). To define forward security formally, the notion of a secure digital signature of [GMR88] is extended in [BM99] to take into account the ability of the adversary to obtain a key by means of a break-in.

Intuitively, in this new model, the forger first conducts an adaptive chosen message attack (cma), requesting signatures on messages of its choice for as many time periods as he desires. Whenever he chooses, he "breaks in": requests the secret key SK_b for the current time period b and then outputs an (alleged) signature on a message M of his choice for a time period $j < b$. The forger is considered to be successful if the signature is valid and the pair (M, j) was not queried during cma.

Formally, let the forger $F = \langle F.\mathsf{cma}, F.\mathsf{forge}\rangle$. For a key pair $(PK, SK_0) \stackrel{R}{\leftarrow} \mathsf{FSIG.key}(k, \ldots, T)$, $F.\mathsf{cma}$, given PK and T, outputs (CM, b), where b is the break-in time period and CM is a set of adaptively chosen message-period pairs (the set of signatures $sign(CM)$ of the current set CM is available to F at all times, including during the construction of CM)[3]. Finally, $F.\mathsf{forge}$ outputs $\langle M, j, sig\rangle \leftarrow F.\mathsf{forge}(CM, sign(CM), SK_b)$. We say that F is successful if $\langle M, j\rangle \notin CM, j < b$, and $\mathsf{FSIG.ver}_{PK}(M, \langle j, sig\rangle) = 1$. (Note: formally, the components of F can communicate all the necessary information, including T and b, via CM.)

Define $\mathbf{Succ}^{\mathrm{fwsig}}(\mathsf{FSIG}[k, T], F)$ to be the probability (over coin tosses of F and FSIG) that F is successful. Let the function $\mathbf{InSec}^{\mathrm{fwsig}}(\mathsf{FSIG}[k, T], t, q_{\mathrm{sig}})$ (the *insecurity* function) be the maximum, over all algorithms F that are restricted to running time t and q_{sig} signature queries, of $\mathbf{Succ}^{\mathrm{fwsig}}(\mathsf{FSIG}[k, T], F)$.

The insecurity function above follows the "concrete security" paradigm and gives us a measure of how secure or insecure the scheme really is. Therefore, we want its value to be as small as possible. Our goal in a security proof will be to find an upper bound for it.

The above definition can be translated to the random oracle model in a standard way [BR93]: by introducing an additional security parameter 1^l, allowing all algorithms the access to the random oracle $H : \{0, 1\}^* \to \{0, 1\}^l$, and considering q_{hash}, the number of queries to the random oracle, as one more parameter for the forger.

2.2 Assumption

We use a variant of the strong RSA assumption (to the best of our knowledge, first introduced independently in [BP97] and [FO97]), which postulates that it is to compute *any* root of a fixed value modulo a composite integer. More precisely, the strong RSA assumption states that it is intractable, given n that is a product of two primes and a value α in Z_n^*, to find $\beta \in Z_n^*$ and $r > 1$ such that $\beta^r = \alpha$.

However, we modify the assumption in two ways. First, we restrict ourselves to the moduli that are products of so-called "safe" primes (a safe prime is one of the form $2q + 1$, where q itself is a prime). Note that, assuming safe primes

[3] Note that the [BM99] definition, which captures what F can do in practice, allows the messages-period pairs to be added to CM only in the order of increasing time periods and without knowledge of any secret keys. However, allowing the forger to construct CM in arbitrary order, and even to obtain SK_b in the middle of the CM construction (so that some messages be constructed by the forger *with* the knowledge of SK_b) would not affect our (and their) results. Similarly, the forger can be allowed to obtain more than one secret key — we only care about the earliest period b for which the secret key is given to the forger. So, the forger may adaptively select some messages which are signed for him, then request some period's secret key; then adaptively select more messages and again request a key, etc.

are frequent, this restriction does not strengthen the assumption. Second, we upperbound the permissible values or r by 2^{l+1}, where l is a security parameter for our scheme (in an implementation, l will be significantly shorter than the length k of the modulus n).

More formally, let A be an algorithm. Consider the following experiment.

Experiment Break-Strong-RSA(k, l, A)
Randomly choose two primes q_1 and q_2 of length $\lceil k/2 \rceil - 1$ each
such that $2q_1 + 1$ and $2q_2 + 1$ are both prime.
$p_1 \leftarrow 2q_1 + 1$; $p_2 \leftarrow 2q_2 + 1$; $n \leftarrow p_1 p_2$
Randomly choose $\alpha \in Z_n^*$.
$(\beta, r) \leftarrow A(n, \alpha)$
`If` $1 < r \leq 2^{l+1}$ `and` $\beta^r \equiv \alpha \pmod{n}$ `then return` 1 `else return` 0

Let $\mathsf{Succ}(A, k, l) = \Pr[Break{-}Strong{-}RSA(k, l, A) = 1]$. Let $\mathbf{InSec}^{\mathrm{SRSA}}(k, l, t)$ be the maximum of $\mathsf{Succ}(A, k, l)$ over all the adversaries A who run in time at most t. Our assumption is that $\mathbf{InSec}^{\mathrm{SRSA}}(k, l, t)$, for t polynomial in k, is negligible in k. The smaller the value of l, of course, the weaker the assumption.

In fact, for a sufficiently small l, our assumption follows from a variant of the fixed-exponent RSA assumption. Namely, assume that there exists a constant ϵ such that, for every r, the probability of computing, in time t, an r-th root of a random integer modulo a k-bit product of two safe primes, is at most 2^{-k^ϵ}. Then, $\mathbf{InSec}^{\mathrm{SRSA}}(k, l, t) < 2^{l+1-k^\epsilon}$, which is negligible if $l = o(k^\epsilon)$.

2.3 Mathematical Tools

The following two simple statements will be helpful later. They were first pointed out by Shamir [Sha83] in the context of generation of pseudorandom sequences based on the RSA function.

Proposition 1. *Let G be a group. Suppose $e_1, e_2 \in Z$ are such that $\gcd(e_1, e_2) = 1$. Given $a, b \in G$ such that and $a^{e_1} = b^{e_2}$, one can compute c such that $c^{e_2} = a$ in $O(\log(e_1 + e_2))$ group and arithmetic operations.*

Proof. Using Euclid's extended gcd algorithm, within $O(\log(e_1 + e_2))$ arithmetic operations compute f_1, f_2, such that $e_1 f_1 + e_2 f_2 = 1$. Compute $c = a^{f_2} b^{f_1}$, with $O(\log(f_1 + f_2)) = O(\log(e_1 + e_2))$ group operations. Then $c^{e_2} = a^{e_2 f_2} b^{e_2 f_1} = a^{e_2 f_2} a^{e_1 f_1} = a$. □

Lemma 1. *Let G be a finite group. Suppose $e_1 \in Z$ and $e_2 \in Z$ are such that $\gcd(e_1, e_2) = g$ and $\gcd(g, |G|) = 1$. Given $a, b \in G$, such that $a^{e_1} = b^{e_2}$, one can compute c such that $c^{e_2/g} = a$ in $O(\log \frac{e_1 + e_2}{g})$ group and arithmetic operations.*

Proof. Since $\gcd(g, |G|) = 1$, $(z^g = 1) \Rightarrow (z = 1)$ for any $z \in G$. Let $e_1' = e_1/g$, $e_2' = e_2/g$. Then $(a^{e_1'}/b^{e_2'})^g = 1$, so $a^{e_1'} = b^{e_2'}$, so we can apply and Proposition 1 to get c such that $c^{e_2'} = a$. □

2.4 The Guillou-Quisquater Signature Scheme

In [GQ88], Guillou and Quisquater propose the following three-round identification scheme, summarized in Figure 1. Let k and l be two security parameters. The prover's secret key consists of a k-bit modulus n (a product of two random primes p_1, p_2), an $(l+1)$-bit exponent e that is relatively prime to $\phi(n) = (p_1 - 1)(p_2 - 1)$, and a random $s \in Z_n^*$. The public key consists of n, e and v where $v \equiv 1/s^e \pmod{n}$.

In the first round, the prover generates a random $r \in Z_n^*$, computes the commitment $y = r^e \pmod{n}$ and sends y to the the verifier. In the second round, the verifier sends a random l-bit challenge σ to the prover. In the third round, the prover computes and sends to the verifier $z = rs^\sigma$. To check, the verifier computes $y' = z^e v^\sigma$ and checks if $y = y'$ (and $y \not\equiv 0 \pmod{n}$).

The scheme's security is based on the assumption that computing roots modulo composite n is infeasible without knowledge of its factors (the precise assumption varies depending on how e is chosen), and can be proven using Lemma 1. Informally, if the prover can answer two different challenges, σ and τ, for the same y, then it can provide z_σ and z_τ such that $z_\sigma^e v^\sigma = z_\tau^e v^\tau$. Hence, $v^{\sigma-\tau} = (z_\sigma/z_\tau)^e$. Note that e is $l+1$-bits long, hence $e > |\sigma - \tau|$, hence $g = \gcd(\sigma - \tau, e) < e$, so $r = e/g > 1$. By Lemma 1, knowing $v, \sigma - \tau, z_\sigma/z_\tau$ and e allows one to efficiently compute the r-th root of v (to apply the lemma, we need to have g relatively prime with the order $\phi(n)$ of the multiplicative group Z_n^*, which is the case by construction, because e is picked to be relatively prime with $\phi(n)$). Thus, the prover must know at least some root of v (in fact, if e is picked to be prime, then the prover must know precisely the e-th root of v, because $g = 1$ and $r = e$). Note that it is crucial to the proof that $e > 2^l$ and e is relatively prime with $\phi(n)$.

The standard transformation of [FS86] can be applied to this identification scheme to come up with the GQ signature scheme, presented in Figure 1. Essentially, the interactive verifier's l-bit challenge σ is now computed using a random oracle (hash function) $H : \{0,1\}^* \to \{0,1\}^L$ applied to the message M and the commitment y.

3 Our Forward-Secure Scheme

3.1 Main ideas for forward security

The main idea for our forward-secure scheme is to combine the GQ scheme with Shamir's observation (Lemma 1). Namely, let $e_1, e_2, \ldots, e_T$ be distinct integers, all greater than 2^l, all pairwise relatively prime and relatively prime with $\phi(n)$. Let $s_1, s_2, \ldots, s_T$ be such that $s_i^{e_i} \equiv 1/v \pmod{n}$ for $1 \le i \le T$. In time period i, the signer will simply use the GQ scheme with the secret key (n, s_i, e_i) and the verifier will use the GQ scheme with the public key (n, v, e_i). Intuitively, this will be forward-secure because of the relative primality of the e_i's: if the forger breaks-in during time period b and learns the e_b-th, e_{b+1}-th, $\ldots$, e_T-th roots of

```
algorithm GQ.key(k, l)
  Generate random ⌈k/2⌉-bit
    primes p_1, p_2
  n ← p_1 p_2
  s ←R Z*_n
  e ←R [2^l, 2^{l+1})
    s.t. gcd(e, φ(n)) = 1
  v ← 1/s^e mod n
  SK ← (n, s, e)
  PK ← (n, v, e)
  return (SK, PK)

algorithm GQ.sign(M, (n, s, e))
  r ←R Z*_n
  y ← r^e mod n
  σ ← H(y, M)
  z ← r s^σ mod n
  return (z, σ)

algorithm GQ.ver(M, (n, v, e), (z, σ))
  if z ≡ 0 (mod n) then return 0
  y' ← z^e v^σ mod n
  if σ = H(y', M) then return 1
                  else return 0
```

Fig. 1. *The GQ Signature Scheme*

v, this will not help it compute e_j-th root of v for $j < b$ (nor, more generally, the r-th root of v, where $r|e_j$).

This idea is quite simple. However, we still need to address the following two issues: (i) how the signer computes the s_i's, and (ii) how both the signer and the verifier obtain the e_i's.

Computing s_i's. Notice that if the signer were required to store all the s_i's, this scheme would require secret storage that is linear in T. However, this problem can be easily resolved. Let $f_i = e_i \cdot e_{i+1} \cdot \ldots \cdot e_T$. Let t_i be such that $t_i^{f_i} \equiv 1/v \pmod{n}$. During the j-th time period, the signer stores s_j and t_{j+1}. At update time, the signer computes $s_{j+1} = t_{j+1}^{f_{j+2}} \bmod n$ and $t_{j+2} = t_{j+1}^{e_{j+1}} \bmod n$. This allows secret storage that is independent of T: only two values modulo n are stored at any time (the f_i and e_i values are not stored—see below). It does, however, require computation linear in T at each update, because of the high cost of computing s_{j+1} from t_{j+1}.

We can reduce the computation at each update to be only logarithmic in T by properly utilizing precomputed powers of t_{j+1}. This will require us, however, to store $1 + \log_2 T$ secrets instead of just two. This optimization concerns only the efficiency of the update algorithm and affects neither the other components of the scheme nor the proof of security, and is therefore presented separately in Section 4.2.

Obtaining e_i's. In order for the scheme to be secure, the e_i's need to be relatively prime with each other[4] and with $\phi(n)$, and greater than 2^l. The signer can therefore generate the e_i's simply as distinct $(l+1)$-bit primes. Of course,

[4] In fact, this requirement can be relaxed. We can allow the e_i's not to be pairwise relatively prime, as long as we redefine f_i as $f_i = \mathrm{lcm}(e_i, e_{i+1}, \ldots, e_T)$, and require that e_i be relatively prime with $\phi(n)$ and $e_i/\gcd(e_i, f_{i+1}) > 2^l$. However, we see no advantages in allowing this more general case; the disadvantage is that the e_i's will

to store all the e_i's would require linear in T (albeit public) storage. However, the signer need only store e_j for the current time period j, and generate anew the other e_i's for $i > j$ during key update. This works as long as the signer uses a deterministic algorithm for generating primes: either pseudorandom search or sequential search from fixed starting points. The fact that e_i's are not stored but rather recomputed each time slows down the update algorithm only (and, as we show in Section 3.3, not by much). Note that the way we currently described the update algorithm, for the update at time period j the signer will need to compute $e_{j+1}, \ldots, e_T$. With the optimization of Section 4.2, however, only at most $\log_2 T$ of the e_i's will need to be computed at each update.

We have not yet addressed the issue of how the verifier gets the e_i's. Of course, it could simply generate them the same way that the signer does during each key update. However, this will slow down verification, which is undesirable. The solution is perhaps surprising: the verifier need not know the "true" e_i's at all! The value of e_j can be simply included by the signer in every signature for time period j. Of course, a forger is under no obligation to include the true e_j. Therefore, to avoid ambiguity, we will denote by e the value included in a signature. It may or may not actually equal e_j.

For the security of the scheme, we require that e satisfy the following requirements:

1. e should be included as an argument to the hash function H, so that the forger cannot decide on e after seeing the challenge σ;
2. e should be greater than 2^l, for the same reasons as in the GQ scheme;
3. e should be relatively prime with $\phi(n)$, for the same reasons as in the GQ scheme; and
4. e should be relatively prime with the $e_b, \ldots, e_T$ (where b is the break-in time period), so that the knowledge of the root of v of degree $e_b \cdot e_{b+1} \cdot \ldots \cdot e_T$ does not help the forger compute any root of v of degree $r|e$.

The first two conditions can be easily enforced by the verifier. The third condition can be enforced by having n be a product of two "safe" primes (primes p_1, p_2 that are of the form $p_i = 2q_i + 1$, where q is prime). Then the verifier simply needs to check that e is odd (then it must be relatively prime with $\phi(n)$—otherwise, it would be divisible by q_1, q_2 or $q_1 q_2$, which would imply that the forger could factor n).

It is the fourth condition that presents difficulties. How can the verifier check the that e is relatively prime with $e_b, \ldots, e_T$ without knowing b and the actual values of $e_b, \ldots, e_T$? We accomplish this by splitting the entire interval between 2^l and 2^{l+1} into T consecutive buckets of size $2^l/T$ each, and having each e_i be a prime from the i-th bucket. Then the verifier knows that the actual values $e_{j+1}, \ldots, e_T$ are all at least $2^l(1 + j/T)$ and prime. Thus, as long as e in the signature for time period j is less than $2^l(1+j/T)$, it is guaranteed to be relatively prime with $e_{j+1}, \ldots, e_T$, and hence with $e_b, \ldots, e_T$ (because $b > j$).

have to be longer to satisfy the last requirement, and thus the scheme will be less efficient.

Thus, to enforce the above four conditions, the verifier needs to check is that e is odd, is between 2^l and $2^l(1+j/T)$ and is included in the hash computation.

3.2 The scheme

Our scheme (denoted IR) based on the above ideas is presented in Figure 2. As in the GQ scheme, let $H : \{0,1\}^* \to \{0,1\}^l$ be a hash function.

3.3 Efficiency

SIGNING AND VERIFYING. The distinghuishing feature of our scheme is the efficiency of the signing and verification algorithms. Both are the same as the already efficient ordinary GQ scheme (verifying has the additional, negligible component of testing whether e is in the right range and odd). Namely, they each take two modular exponentiations, one modular multiplication and an application of H, for a total time of $O(k^2 l)$ plus the time required to evaluate H. (Note that, just like the GQ scheme, one of the two modular exponentiations for signing can be done off-line, before the message is known; also, one of the two modular exponentiations for verifying is of a fixed base v, and can benefit from precomputation.)

KEY GENERATION. We need to make strong assumptions on the distributions of primes in order to estimate efficiency of key generation. First, we assume that at least one in $O(k)$ $\lceil k/2 \rceil$-bit numbers is a prime, and that at least one in $O(k)$ of those is of the form $2q+1$, where q is prime. Then, generating n takes $O(k^2)$ primality tests. Each primality test can be done in $O(k^3)$ bit operations [BS96]. Thus, the modulus n is generated in $O(k^5)$ bit operations (a factor k slower than an RSA modulus, because of the need for safe primes). Similarly, we will assume that at least one in $O(l)$ integers in each bucket $[2^l(1+(i-1)/T), 2^l(1+i/T))$ is a prime, so generating each e_i takes $O(l^4)$ bit operations.

In addition to generating n and the e_i's, key generation needs to compute the product of the e_i's modulo $\phi(n)$, which takes $O(Tkl)$ bit operations, and three modular exponentiations, each taking $O(k^2 l)$ bit operations. Therefore, key generation takes $O(k^5 + l^4 T + k^2 l + klT))$ bit operations.

Note that, similarly to the GQ scheme, n and e_i's may be shared among users if n is generated by a trusted party, because each user need not know the factors of n. Each user can simply generate its own t_1 and v.

KEY UPDATE. Key update cannot multiply all the relevant e_i's modulo $\phi(n)$, because $\phi(n)$ is not available (otherwise, the scheme would not be forward-secure). Therefore, it has to perform $O(T)$ modular exponentiations separately, in addition to regenerating all the e_i's. Thus, it takes $O(k^2 lT + l^4 T)$ bit operations.

Note that the $l^4 T$ component is present in the running time for the update algorithm because of the need to regenerate the e_i's each time. However, for practical values of l (on the order of 100) and k (on the order of 1000), $l^4 T$ is roughly the same as $k^2 lT$, so this only slows down the key update algorithm by

algorithm IR.key(k, l, T)
Generate random $(\lceil k/2 \rceil - 1)$-bit primes q_1, q_2 s.t. $p_i = 2q_i + 1$ are both prime
$n \leftarrow p_1 p_2$
$t_1 \stackrel{R}{\leftarrow} Z_n^*$
Generate primes e_i s.t. $2^l(1 + (i-1)/T) \le e_i < 2^l(1 + i/T)$ for $i = 1, 2, \ldots, T$.
(This generation is done either deterministically or using a small seed *seed* and H as a pseudorandom function.)
$f_2 \leftarrow e_2 \cdot \ldots \cdot e_T \bmod \phi(n)$, where $\phi(n) = 4q_1 q_2$
$s_1 \leftarrow t_1^{f_2} \bmod n$
$v \leftarrow 1/s_1^{e_1} \bmod n$
$t_2 \leftarrow t_1^{e_1} \bmod n$
$SK_1 \leftarrow (1, T, n, s_1, t_2, e_1, seed)$
$PK \leftarrow (n, v, T)$
return (SK_1, PK)

---

algorithm IR.update(SK_j)
Let $SK_j = (j, T, n, s_j, t_{j+1}, e_j, seed)$
if $j = T$ **then return** ϵ
Regenerate $e_{j+1}, \ldots, e_T$ using *seed*
$s_{j+1} \leftarrow t_{j+1}^{e_{j+2} \cdot \ldots \cdot e_T} \bmod n$; $t_{j+2} \leftarrow t_{j+1}^{e_{j+1}} \bmod n$
return $SK_{j+1} = (j+1, T, n, s_{j+1}, t_{j+2}, e_{j+1}, seed)$

---

algorithm IR.sign(SK_j, M)
Let $SK_j = (j, T, n, s_j, t_{j+1}, e_j, seed)$
$r \stackrel{R}{\leftarrow} Z_n^*$
$y \leftarrow r^{e_j} \bmod n$
$\sigma \leftarrow H(j, e_j, y, M)$
$z \leftarrow r s^{\sigma} \bmod n$
return (z, σ, j, e_j)

---

algorithm IR.ver$(PK, M, (z, \sigma, j, e))$
Let $PK = (n, v)$
if $e \ge 2^l(1 + j/T)$ **or** $e < 2^l$ **or** e is even **then return** 0
if $z \equiv 0 \pmod{n}$ **then return** 0
$y' \leftarrow z^e v^{\sigma} \bmod n$
if $\sigma = H(j, e, y', M)$ **then return** 1 **else return** 0

Fig. 2. *Our forward-secure signature scheme (without efficiency improvements)*

a small constant factor. Moreover, in Section 4.1 we show how to reduce the l^4T component in both key generation and update to $(l^2 + \log^4 T)T$ (at a very slight expense to signing and verifying).

Finally, as shown in Section 4.2, if we are willing to increase secret storage from $2k$ bits (for s_j and t_{j+1}) to $(1+\log_2 T)k$ bits, then we can replace the factor of T in the cost of update by the factor of $\log_2 T$, to get update at the cost of $O((l^4 + k^2 l)\log T)$ (or, if optimization of Section 4.1 is additionally applied, $O((k^2 l + l^2 + \log^4 T)\log T))$.

Sizes. All the key and signature sizes are comparable to those in the ordinary GQ scheme.

The public key has $l+1$ fewer bits than the GQ public key, and the signatures have $l+1$ more bits, because e is included in the signature rather than in the public key. In addition, both the public key and the signature have $\log_2 T$ more bits in order to accommodate T in the public key and the current time period in the signature (this is necessary in any forward-secure scheme). Thus, the total public key length is $2k + \log_2 T$ bits, and signature length is $k + 2l + 1 + \log_2 T$ bits. Optimization of Section 4.1 shortens the signatures slightly, replacing $l+1$ of the signature bits with about $\log_2 T$ bits.

The secret key is $k + 2\log_2 T + |seed|$ bits longer than in the GQ scheme in order to accommodate the current time period j, the total time periods T, the value t_{j+1} necessary to compute future keys and the seed necessary to regenerate the e_i's for $i > j$. Thus, the total secret key length is $3k + l + 1 + |seed| + 2\log_2 T$ bits (note that only $2k$ of these bits need to be kept secret). If the optimization of Section 4.2 is used, then the secret contains an additional $k(\log_2 T - 1)$ bits, all of which need to be kept secret.

3.4 Security

The exact security of our scheme (in the random oracle model) is close to the exact security of the schemes of [BM99,AR00]. The proof is also similar: it closely follows the one in [AR00], combining ideas from [PS96,BM99,MR99].

First, we state the following theorem that will allow us to upper-bound the insecurity function. The full proof of the theorem is very similar to the one in [AR00] and is contained in Appendix A.

Theorem 1. *Given a forger F for $\mathsf{IR}[k,l,T]$ that runs in time at most t, asking q_{hash} hash queries and q_{sig} signing queries, such that $\mathbf{Succ}^{\text{fwsig}}(\mathsf{IR}[k,l,T],F) \geq \varepsilon$, we can construct an algorithm A that, on input n (a product of two safe primes), $\alpha \in Z_n^*$ and l, runs in time t' and outputs (β, r) such that $1 < r \leq 2^{l+1}$ and $\beta^r \equiv \alpha \pmod n$ with probability ε', where*

$$t' = 2t + O(lT(l^2T^2 + k^2))$$
$$\varepsilon' = \frac{\left(\varepsilon - 2^{2-k} q_{\text{sig}}(q_{\text{hash}} + 1)\right)^2}{T^2(q_{\text{hash}} + 1)} - \frac{\varepsilon - 2^{2-k} q_{\text{sig}}(q_{\text{hash}} + 1)}{2^l T}.$$

Proof Outline. A will use F as a subroutine. (Note that A gets to provide the public key for F and to answer its signing and hashing queries.) A bases the public key v on α as follows: it randomly guesses j between 1 and T, hoping that F's eventual forgery will be for the j-th time period. It then generates $e_1, \ldots, e_T$ just like the real signer, sets $t_{j+1} = \alpha$ and computes v as $v = 1/t_{j+1}^{f_{j+1}} \bmod n$, where, as above, $f_{j+1} = e_{j+1} \cdot \ldots \cdot e_T$.

Then A runs F. Answering F's hash and signature queries is easy, because A fully controls the random oracle H. If A's guess for j was correct, and F indeed will output a forgery for the j-th time period, then F's break-in query will be for the secret of a time period $b > j$. A can compute the answer as follows: $t_{b+1} = t_{j+1}^{f_{j+1}/f_b} = \alpha^{e_{j_1} \cdot \ldots \cdot e_b}$ and $s_b = t_b^{f_{b+1}} = \alpha^{e_{j_1} \cdot \ldots \cdot e_{b-1} \cdot e_{b+1} \cdot \ldots \cdot e_T}$ (the other components of SK_b are not secret, anyway). Suppose A's guess was correct, and in the end F outputs a signature (z, σ, j, e) on some message M. We will assume that F asked a hash query on (j, e, y, M) where $y = z^e v^\sigma \bmod n$ (F can always be modified to do so.)

Then, A runs F the second time with the same random tape, giving the same answers to all the oracle queries before the query (j, e, y, M). For (j, e, y, M), A gives a new answer τ. If F again forges a signature (z', τ, j, e) using the same hash query, we will have that $y \equiv z^e v^\sigma \equiv {z'}^e v^\tau \pmod{n}$, so $(z/z')^e \equiv v^{\tau-\sigma} \equiv \alpha^{f_{j+1}(\sigma-\tau)} \pmod{n}$. Note that because e is guaranteed to be relatively prime with f_{j+1}, and $\sigma - \tau$ has at least one fewer bit than e, $\gcd(f_{j+1}(\sigma - \tau), e) = \gcd(\sigma - \tau, e) < e$ (as long as $\sigma \neq \tau$). Thus, $r = e/\gcd(f_{j+1}(\tau - \sigma), e) > 1$ and, by Lemma 1, A will be able to efficiently compute the r-th root of α.

Please refer to Appendix A for further details. □

This allows us to state the following theorem about the insecurity function of our scheme.

Theorem 2. *For any t, q_{sig}, and q_{hash},*

$$\mathbf{InSec}^{\text{fwsig}}(\mathsf{IR}[k,l,T]; t, q_{\text{sig}}, q_{\text{hash}}) \leq$$
$$T\sqrt{(q_{\text{hash}}+1)\mathbf{InSec}^{\text{SRSA}}(k,l,t')} + 2^{-l+1}T(q_{\text{hash}}+1) + 2^{2-k}q_{\text{sig}}(q_{\text{hash}}+1) \ ,$$

where $t' = 2t + O(lT(l^2T^2 + k^2))$.

Proof. To compute the insecurity function, simply solve for $(\varepsilon - 2^{2-k}q_{\text{sig}}(q_{\text{hash}} + 1))/T$ the quadratic equation in Theorem 1 that expresses ε' in terms of ε to get

$$\begin{aligned}
&(\varepsilon - 2^{2-k}q_{\text{sig}}(q_{\text{hash}}+1))/T \\
&\qquad = 2^{-l}(q_{\text{hash}}+1) + \sqrt{2^{-2l}(q_{\text{hash}}+1)^2 + \varepsilon'(q_{\text{hash}}+1)} \\
&\qquad \leq 2^{-l}(q_{\text{hash}}+1) + \sqrt{2^{-2l}(q_{\text{hash}}+1)^2} + \sqrt{\varepsilon'(q_{\text{hash}}+1)} \\
&\qquad = 2^{-l+1}(q_{\text{hash}}+1) + \sqrt{\varepsilon'(q_{\text{hash}}+1)},
\end{aligned}$$

and then solve the resulting inequality for ε. □

4 Further Improving Efficiency

4.1 Finding the e_i's faster

Finding e_i's takes time because they need to be $l+1$-bit primes. If we were able to use small primes instead, we could search significantly faster, both because small primes are more frequent and because primality tests are faster for shorter lengths.[5]

We cannot use small primes directly because, as already pointed out, the e_i's must have at least $l+1$ bits. However, we can use powers of small primes that are at least $l+1$ bits. That is, we let ϵ_i be a small prime, $\pi(\epsilon_i)$ be such that $\epsilon_i^{\pi(\epsilon_i)} > 2^l$ and $e_i = \epsilon_i^{\pi(\epsilon_i)}$. As long as π is a deterministic function of its input ϵ (for example, $\pi(\epsilon) = l/\lfloor \log_2 \epsilon \rfloor$), we can replace e in the signature by ϵ, and have the verification algorithm compute $e = \epsilon^{\pi(\epsilon)}$.

Of course, the verification algorithm still needs to ensure that e is relatively prime to $\phi(n)$ and to $e_b, \ldots, e_T$. This is accomplished essentially the same way as before: we divide a space of small integers into T consecutive buckets of some size S each, and have each ϵ_i come from the i-th bucket: $\epsilon_i \in [(i-1)S, iS)$. Then, when verifying a signature for time period j, it will suffice to check that ϵ is odd and comes from a bucket no greater than the j-th: $\epsilon < jS$. It will be then relatively prime to $\epsilon_b, \ldots, \epsilon_T$, and therefore $e = \epsilon^{\pi(\epsilon)}$ will be relatively prime to $e_b, \ldots, e_T$.

When we used large primes, we simply partitioned the space of $(l+1)$-bit integers into large buckets, of size $2^l/T$ each. We could have used smaller buckets, but this offered no advantages. However, now that we are using small primes, it is advantageous to make the bucket size S as small as possible, so that even the largest prime (about TS) is still small.

Thus, to see how much this optimization speeds up the search for the e_i's, we need to upper-bound S. S needs to be picked so that there is at least one prime in each interval $[(i-1)S, iS)$ for $1 \le i \le T$. It is reasonable to conjecture that the distance between two consecutive primes P_n and P_{n+1} is at most $(\ln^2 P_n)$ [BS96]. Therefore, because the largest prime we are looking for is smaller than TS, S should be such that $S > \ln^2 TS$. It is easy to see that $S = 4\ln^2 T$ will work for $T \ge 75$. (As a practical matter, computation shows that, for any reasonable value of T, the value of S will be quite small: $S = 34$ will work for $T = 1000$, because the largest gap between the first 1000 primes is 34; by the same reasoning, $S = 72$ will work for $T = 10^4$, $S = 114$ will work for $T = 10^5$, and $S = 154$ will work for $T = 10^6$.) Thus, the ϵ_i's are all less than $4T\ln^2 T$, and therefore the size of each ϵ_i is $O(\log T)$ bits. Thus, finding and testing the primality of the ϵ_i's and then computing the e_i's takes $O(T(\log^4 T + l^2))$ time, as opposed to $O(Tl^4)$ without this optimization.

The resulting scheme will slightly increase verification time: the verifier needs to compute e from ϵ. This takes time $O(l^2)$ (exponentiating any quantity to

[5] In fact, when a table of small primes is readily available (as it often is for reasonably small T), no searching or primality tests are required at all.

obtain an $(l+1)$-bit quantity takes time $O(l^2)$), which is lower order than $O(k^2 l)$ verification time. Moreover, it will be impossible to get e_i to be exactly $l+1$ bits (it will be, on average, about $l + (\log_2 T)/2$ bits). This will slow down both verification and signing, albeit by small amounts. Therefore, whether to use the optimization in practice depends on the relative importance of the speeds of signing and verifying vs. the speeds of key generation and update.

4.2 Optimizing key update

The key update in our scheme requires computing s_i such that $s_i^{e_i} \equiv 1/v \bmod n$. Knowledge of s_{i-1}, such that $s_{i-1}^{e_{i-1}} \equiv 1/v \bmod n$, does not help, because e_i and e_{i-1} are relatively prime. The easiest way to compute s_i requires knowledge of $\phi(n)$: $s_i \leftarrow 1/v^{1/e_i \bmod \phi(n)} \bmod n$. However, the signer cannot store $\phi(n)$—otherwise the forger would obtain it during a break-in, and thus be able to factor n and produce the past periods' secrets (and signatures). The value of $\phi(n)$ can be used only during the initial key generations stage, after which it should be securely deleted.

To enable generation of current and future s_i's without compromising the past ones, we had defined (in Section 3) a secret t_i for time period i, from which it was possible to derive all future periods' secrets $s_{j \geq i}$. The update of t_i to t_{i+1} can be implemented efficiently (1 exponentiation). However, in this approach the computation of each s_i from t_i requires $\Theta(T-i)$ exponentiations. This computation can be reduced dramatically if the storage is increased slightly.

Specifically, in this section we demonstrate how replacing the single secret t_i with $\log_2 T$ secrets can reduce the complexity of the update algorithm to only $\log_2 T$ exponentiations.

Abstracting the Problem. Consider all subsets of $Z_T = \{1, 2, \ldots, T\}$. Let each such subset S correspond to the secret value $t_S = t_1^{\prod_{i \notin S} e_i}$. For example, t_1 corresponds to Z_T, t_i corresponds to $\{i, i+1, \ldots, T\}$, v^{-1} corresponds to the empty set, and each s_i corresponds to the singleton set $\{i\}$. Raising some secret value t_S to power e_i corresponds to dropping i from S.

Thus, instead of secrets and the exponentiation operation, we can consider sets and the operation of removing an element. Our problem, then, can be reformulated as follows: design an algorithm that, given Z_T, outputs (one-by-one, in order) the singleton sets $\{i\}$ for $1 \leq i \leq T$. The only way to create new sets is to remove elements from known sets. The algorithm should minimize the number of element-removal operations (because they correspond to the expensive exponentiation operations).

Fairly elementary analysis quickly demonstrates that the most efficient solution for this problem (at least for T that is a power of 2) is the following divide-and-conquer algorithm:

Input: An ordered non-empty set A.
Output: Singleton sets $\{x\}$, for $x \in A$, in order.
Steps: If A has one element, output A and return.

Remove the second half of A's elements to get B.
Recurse on B.
Remove the first half of A's elements to get C.
Recurse on C.

This algorithm takes exactly $T \log_2 T$ element-removal operations to output all the singletons. Moreover, the recursion depth is $1 + \log_2 T$, so only $1 + \log_2 T$ sets need to be stored at any time (each set is just a consecutive interval, so the bookkeeping about what each set actually contains is simple).

This recursive algorithm can essentially be the update algorithm for our scheme: at every call to update, we run the recursive algorithm a little further, until it produces the next output. We then stop the recursive algorithm, save its stack (we need to save only $\log_2 T$ secrets, because the remaining one is the output of the algorithm), and run it again at the next call to update. A little more care needs to be taken to ensure forward security: none of the sets stored at time period i should contain elements less than i. This can be done by simply removing i from all sets that still contain in (and that are still needed) during the i-th update. The total amount of work still does not change.

Because there are T calls to update (if we include the initial key generation), the amortized amount of work per update is exactly $\log_2 T$ exponentiations. However, some updates will be more expensive than others, and update will still cost $\Theta(T)$ exponentiations in the worst case. We thus want to improve the worst-case running time of our solution without increasing the (already optimal) total running time. This can be done through pebbling techniques, described below.

PEBBLING. Let each subset of Z_T correspond to a node in a graph. Connect two sets by a directed edge if the destination can be obtained from the source by dropping a single element. The resulting graph is the T-dimensional hypercube, with directions on the edges (going from higher-weight nodes to lower-weight nodes). We can traverse the graph in the direction given by the edges. We start at the node corresponding to Z_T, and need to get to all the nodes corresponding to the singleton sets $\{i\}$.

One way to accomplish this task is given by the above recursive algorithm, which has the minimal total number of steps. However, we would like to minimize not only the total number of steps, but also the number of steps taken *between* any two "consecutive" nodes $\{i\}$ and $\{i+1\}$, while keeping the memory usage low. We will do this by properly arranging different branches of the recursive algorithm to run in parallel.

To help visualize the algorithm, we will represent each set stored as a pebble at the corresponding node in a graph. Then removing an element from a set corresponds to moving the corresponding pebble down the corresponding directed edge. The original set may be preserved, in which case a "clone" of a pebble is left at the original node, or it may be discarded, in which case no such clone is left. Our goal can be reformulated as follows in terms of pebbles: find a pebbling strategy that, starting at the node Z_T, reaches every node $\{i\}$ in order, while minimizing the number of pebbles used at any given time (this corresponds to total secret storage needed), the total number of pebble moves (this corresponds to total number of exponentiations needed), and the number of pebble moves between any two consecutive hits of a singleton (this corresponds to the worst-case cost of the update algorithm).

THE PEBBLING ALGORITHM. We shall assume that $T > 1$ is a power of 2. The following strategy uses at most $1 + \log_2 T$ pebbles, takes $T \log_2 T$ total moves (which is the minimum possible), and requires at most $\log_2 T$ moves per update.

Each pebble has the following information associated with it:

1. its current position, represented by a set $P \subseteq Z_T$ (P will always be a set of consecutive integers $\{P_{\min}, \ldots, P_{\max}\}$);
2. its "responsibility," represented by a set $R \subseteq P$ (R will also always be a set of consecutive integers $\{R_{\min}, \ldots, R_{\max}\}$; moreover $|R|$ will always be a power of 2).

Each pebble's goal is to ensure that it (together with its clones, their clones, etc.) reaches every singleton in its set P. If $R \subsetneq P$, then the pebble can move towards this goal by removing an element from P. If, however, $R = P$, then the pebble has to clone (unless $|P| = |R| = 1$, in which case it has reached its singleton, and can be removed from the graph). Namely, it creates a new pebble with the same P, and responsibility set R' containing only the second half of R. It then changes its own R to $R - R'$ (thus dividing its responsibility evenly between itself and its clone). Now both the pebble and the clone can move towards their disjoint sets of singletons.

We start with a single pebble with $P = R = Z_T$. The above rules for moving and cloning ensure that the combined moves of all the pebbles will be the same as in the recursive algorithm. Thus, the steps of the pebbles are already determined. We now have to specify the timing rules: namely, when the pebbles take their steps. A careful specification is important: if a pebble moves too fast, then it can produce more clones than necessary, thus increasing the total memory; if a pebble moves too slowly, then it may take longer to reach its destination singletons, thus increasing the worst-case cost of update.

In order to specify the timing rules, we will imagine having a clock. The clock "ticks" consecutive integer values, starting with $-T/2+1$. After each clock tick, each pebble will decide whether to move and, if so, for how many moves, as follows:

1. The original pebble always makes two moves per clock tick, until it reaches the singleton $\{1\}$. After reaching the singleton it stops, and then removes itself from the graph on the next clock tick.
2. After a new pebble is cloned with responsibility set R, it stays still for $\lceil |R|/2 \rceil$ clock ticks. After $\lceil |R|/2 \rceil$-th clock-tick following its birth, it starts moving at one move per clock tick. After $|R|$ such moves, it starts moving a two moves per clock tick, until it reaches its leftmost singleton. After reaching the singleton it stops, and then removes itself from the graph on the next clock tick.

We remark that the above rules may seem a bit complex. Indeed, simpler rules can be envisioned: for example, allowing each pebble at most one move per clock tick, and specifying that each pebbles moves following a given clock tick only if it absolutely has to move in order to reach its leftmost singleton on time. However, this set of rules will require $(\log_2 T) - 2$ pebbles (even though at most $\log_2 T$ of them will be moving at any given time). Having pebbles move at variable speeds allows us to delay their cloning, and thus reduces the total number of pebbles, as shown by the following theorem.

Theorem 3. *Suppose $T > 1$ is a power of two. If i is the value most recently ticked by the clock, then the total number of pebbles under the above rules never exceeds $1 + \lfloor \log_2(T - i) \rfloor$ (if $i \geq 0$) or $(\log_2 T) - \lfloor \log_2 -i \rfloor$ (if $-T < i < 0$). The number of moves occurring immediately following the clock tick i also never exceeds this quantity. For each i, $1 \leq i \leq T$, a pebble reaches the singleton $i + 1$ immediately before the clock ticks the value $i + 1$, and is removed before the clock ticks $i + 2$.*

Proof. The proof is by induction on $\log_2 T$.

For $T = 2$, we start with a single pebble with $P = R = \{1, 2\}$. After the clock ticks 0, this pebble clones the pebble with $R' = 2$, and itself moves to $P = \{1\}$. The clone waits for one clock tick and then, after the clock ticks 1, the clone moves to $P = \{2\}$.

Suppose the statement is true for some T that is a power of two. We will now prove it for $T' = 2T$. After clock tick $-T + 1$, we have two pebbles: one responsible for $\{1, \ldots, T\}$, and the other responsible for $\{T + 1, \ldots, 2T\}$. For the next $T/2 - 1$ clock ticks, the first pebble will move at two steps per tick, and the second one will stay put (thus, the number of moves does not exceed the number of pebbles). After the clock ticks $-T/2$, the first pebble will arrive at position $P = \{1, \ldots, T\}$. Thus, starting at $t = -T + 1$, the inductive hypothesis applies to the all the pebbles that will cover the first half of the singletons: there is a single pebble until $t = -T/2 + 1$ and it is in position $P = \{1, \ldots, T\}$ after clock tick $-T/2 + 1$.

The second pebble will reach the position $P' = \{2, \ldots, T\}$ after the clock ticks $T/2$. Thus, again, after the clock ticks 1, the inductive hypothesis applies to all the pebbles that will cover the second half of the singletons, except that time is shifted forward by T. That is, if $1 \leq i < T$, then the number of pebbles in the second half does not exceed $(\log_2 T) - \lfloor \log_2(T - i) \rfloor$, and if $t \geq T$, then the number of pebbles in the second half does not exceed $1 + \lfloor \log_2(2T - i) \rfloor$.

The key to finishing the proof is to realize that the first half will lose a pebble just as the second half gains one. To be precise, we can consider the following four cases.

- For $-T < i < 0$, we have $(\log_2 T) - \lfloor \log_2 -i \rfloor$ pebbles in the first half (by the inductive hypothesis), and one pebble in the second half, so we have a total of $(\log_2 2T) - \lfloor \log_2 -i \rfloor$ pebbles, as required.
- For $i = 0$, we have $1 + \log_2 T = \log_2 2T$ pebbles in the first half (by the inductive hypothesis), and one pebble in the second half, for a total of $1 + \log_2 2T$ pebbles, as required.

– For $0 < i \leq T$, we have $1 + \lfloor \log_2(T - i) \rfloor$ pebbles in the first half and $(\log_2 T) - \lfloor \log_2(T - i) \rfloor$ pebbles in the second half (both by the inductive hypothesis), for a total of $1 + \log_2 T = 1 + \lfloor \log_2(2T - i) \rfloor$ pebbles, as required.
– For $i > T$, we have no pebbles in the first half and $\lfloor \log_2(2T - i) \rfloor$ pebbles in the second half (by the inductive hypothesis), as required.

It is easy to see that in each of the above four cases, the number of moves does not exceed the number of pebbles (because for every pebble moving at two steps per clock tick, there exists a pebble that is standing still—namely, its most recent clone). □

Security. It is, of course, crucial to ensure that the above changes to the update algorithm do not compromise the security of our scheme. It suffices to prove that every secret stored following the clock tick i can be derived in polynomial time from t_{i+1}. In other words, it suffices to prove that, following the clock tick i, no pebble's position P satisfies $i \in P$. This can be easily done by induction, as long as each pebble moves towards its goal by removing the *smallest* possible element from its position P (the inductive step is proved as follows: if $2T$ is the total number of time periods, then the single pebble responsible for the second half of the singletons will have removed $\{1, \ldots, T/2\}$ from its position following the clock tick 1, and will have removed $\{1, \ldots, T\}$ following the clock tick $T/2 + 1$).

Acknowledgements

We thank Anna Lysyanskaya and Silvio Micali for helpful discussions about our complexity assumptions; Ron Rivest for sharing his insights on pebbling algorithms; and the anonymous referees for helpful comments.

References

[And97] Ross Anderson. Invited lecture. Fourth Annual Conference on Computer and Communications Security, ACM, 1997.

[AR00] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In *Advances in Cryptology—ASIACRYPT 2000*, Springer-Verlag 2000. Full version available from the Cryptology ePrint Archive, record 2000/002, `http://eprint.iacr.org/`.

[BM99] Mihir Bellare and Sara Miner. A forward-secure digital signature scheme. In *Advances in Cryptology—CRYPTO '99*, Springer-Verlag, 1999. Revised version is available from `http://www.cs.ucsd.edu/ mihir/`.

[BP97] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology—EUROCRYPT 97*, Springer-Verlag, 1997.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communication Security*, pages 62–73, November 1993. Revised version appears in `http://www-cse.ucsd.edu/users/mihir/papers/crypto-papers.html`.

[BS96] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory.* MIT Press, Cambridge, MA, 1996.

[FO97] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 17–21 August 1997.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology—CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer-Verlag, 1987, 11–15 August 1986.

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

[Gol88] Shafi Goldwasser, editor. *Advances in Cryptology—CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990, 21–25 August 1988.

[GQ88] Louis Claude Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In Goldwasser [Gol88], pages 216–231.

[Kra00] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *Seventh ACM Conference on Computer and Communication Security.* ACM, November 1–4 2000.

[MI] Silvio Micali and Gene Itkis. Private Communication.

[Mic94] Silvio Micali. A secure and efficient digital signature algorithm. Technical Report MIT/LCS/TM-501, Massachusetts Institute of Technology, Cambridge, MA, March 1994.

[MR99] Silvio Micali and Leonid Reyzin. Improving the exact security of Fiat-Shamir signature schemes. In R. Baumgart, editor, *Secure Networking — CQRE [Secure] '99*, volume 1740 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 1999.

[OO88] Kazuo Ohta and Tatsuaki Okamoto. A modification of the Fiat-Shamir scheme. In Goldwasser [Gol88], pages 232–243.

[OS90] H. Ong and Claus P. Schnorr. Fast signature generation with a Fiat Shamir-like scheme. In I. B. Damgård, editor, *Advances in Cryptology—EUROCRYPT 90*, volume 473 of *Lecture Notes in Computer Science*, pages 432–440. Springer-Verlag, 1991, 21–24 May 1990.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 387–398. Springer-Verlag, 12–16 May 1996.

[Sha83] Adi Shamir. On the generation of cryptographically strong pseudorandom sequences. *ACM Transactions on Computer Systems*, 1(1):38–44, February 1983.

A Details of the Proof of Theorem 1

First, we assume that if F outputs (z, σ, j, e) as a forgery, then the hashing oracle has been queried on (j, e, y, M), where $y = z^e v^\sigma \bmod n$ (any adversary can be modified to do that; this may raise the number of hash queries to $q_{\text{hash}} + 1$.)

We will also assume that F performs the necessary bookkeeping and does not ask the same hash query twice.[6] Note that F may ask the same signature query twice, because the answers will most likely be different.

Recall that A's job, given α and n, is to find (with F's help) β and $r > 1$ such that $\beta^r \equiv \alpha \pmod{n}$. First, A has to guess the time period for which F will output the forgery: it randomly selects j, $1 < j \le T$ (sometimes A may also succeed if the forgery is for a time period $i < j$, but this not necessary for our argument). A then generates $e_1, \ldots, e_T$ just like the real signer, sets $t_{j+1} = \alpha$ and computes v as $v = 1/t_{j+1}^{f_{j+1}} \bmod n$, where, as above, $f_{j+1} = e_{j+1} \cdot \ldots \cdot e_T$.

A then comes up with a random tape for F, remembers it, and runs F on that tape and the input public key (n, v, T). If F breaks in at time period b, then A can provide F with the secret key as long as $b > j$: knowing t_{j+1} will allow A to compute s_b and t_{b+1}. If $b \le j$, then A aborts (because, in particular, F's forgery cannot be for time period j in that case).

To answer F's signature and hash queries, A maintains two tables: a signature query table and a hash query table.

Signature queries can be answered almost at random, because A controls the hash oracle. In order to answer a signature query number s on a message M_s during time period j_s, A selects a random $z_s \in Z_n^*$ and $\sigma_s \in \{0,1\}^l$, computes $y_s = {z_s}^{e_{j_s}} v^{\sigma_s}$, and checks its signature query table to see if a signature query on M_s during time period j_s has already been asked and if y_s used in answering it. If so, A changes z_s and σ_s to the z and σ that were used in answering that query. Then A adds the entry $(s, j_s, e_{j_s}, y_s, \sigma_s, z_s, M_s)$ to its signature query table and outputs $(z_s, \sigma_s, j_s, e_{j_s})$.

Hash queries are also answered at random. To answer the t-th hash query (j'_t, e'_t, y'_t, M'_t), A first checks its signature query table to see if there is an entry $(s, j_s, e_{j_s}, y_s, \sigma_s, z_s, M_s)$ such that $(j_s, e_{j_s}, y_s, M_s) = (j'_t, e'_t, y'_t, M'_t)$. If so, it just outputs σ_s. Otherwise, it picks a random $\sigma'_t \in \{0,1\}^l$, records in its hash query table the tuple $(t, y'_t, M'_t, j'_t, e'_t, \sigma'_t)$ and outputs σ'_t.

Assume now the break-in query occurs during time period $b > j$, and the valid forgery (z, σ, i, e) is output for a time period $i \le j$ (if not, or if no valid forgery is output, A fails). Let $y = z^e v^\sigma$. Because we modified F to first ask a hash query on (i, e, y, M), we have that, for some h, $(h, y, M, i, e, \sigma) = (h, y'_h, M'_h, j'_h, e'_h, \sigma'_h)$ in the hash query table (it can't come from the signature query table, because F is not allowed to forge a signature on a message for which it asked a signature query). A finds such an h in its table and remembers it.

A now resets F with the same random tape as the first time, and runs it again, giving the exact same answers to all F's queries before the h-th hash query (it can do so because it has all the answers recorded in the tables). Note that this means that F will be asking the same h-th hash query (i, e, y, M) as the first time. As soon as F asks the h-th hash query, however, A stops giving the answers from the tables and comes up with new answers at random, in the

[6] This may slightly increase the running time of F, but we will ignore costs of simple table look-up for the purposes of this analysis.

same manner as the first time. Let τ be the new answer given to the h-th hash query, and assume $\tau \neq \sigma$.

Assume again the break-in query occurs during time period $b > j$, and the valid forgery (z', σ', i', e') is output for a time period $i' \leq j$. A again computes $y' = z'^{e'} v^{sigma'}$; by the same reasoning as before, F had to ask a hash query on (i', e', y', M'). Let h' be the number of that query. A finds h' and fails if $h' \neq h$. If, however, $h' = h$, then $(i, e, y, M) = (i', e', y', M')$, simply because the h-th hash query had to be the same in both runs of F. Also then $\sigma' = \tau$. Therefore, $z^e v^\sigma \equiv z'^e v^\tau$, so $(z/z')^e \equiv v^{\tau - \sigma} \equiv \alpha^{f_{j+1}(\sigma - \tau)} \pmod N$.

Note that because e is guaranteed to be relatively prime with f_{j+1} (as long as $i \leq j$), and $\sigma - \tau$ has at least one fewer bit than e, $\gcd(f_{j+1}(\sigma - \tau), e) = \gcd(\sigma - \tau, e) < e$ (as long as $\sigma \neq \tau$). Thus, $r = e/\gcd(f_{j+1}(\sigma - \tau), e) > 1$ and, by Lemma 1, A will be able to efficiently compute the r-th root of α.

Running Time Analysis. A runs F twice. Preparing the public key and answering hashing and signing queries takes A no longer than it would take the real oracles. To find the hashing query corresponding to the forgery and to apply Lemma 1 takes $O(lT(l^2T^2 + k^2))$ bit operations.

Probability Analysis. We will need the following lemma in our analysis.

Lemma 2. *Let $a_1, a_2, \ldots, a_\lambda$ be real numbers. Let $a = \sum_{\mu=1}^{\lambda} a_\mu$, and let $s = \sum_{\mu=1}^{\lambda} a_\mu^2$. Then $s \geq \frac{a^2}{\lambda}$.*

Proof. Let $b = a/\lambda$ and $b_\mu = b - a_\mu$. Then $\sum_{\mu=1}^{\lambda} b_\mu = \lambda b - \sum_{\mu=1}^{\lambda} a_\mu = 0$. Hence $\sum_{\mu=1}^{\lambda} {a_\mu}^2 = \sum_{\mu=1}^{\lambda} (b - b_\mu)^2 = \lambda b^2 - 2b \sum_{\mu=1}^{\lambda} b_\mu + \sum_{\mu=1}^{\lambda} b_\mu^2 \geq \lambda b^2 = \frac{a^2}{\lambda}$. □

First, consider the probability that A's answers to F's oracle queries are distributed as those of the true oracles that F expects. This is the case unless, for some signature query, the hash value that A needs to define has already been defined through a previous answer to a hash query (call this "A's failure to pretend"). Because z is picked at random from Z_n*, $z^e v^\sigma$ is a random element of Z_n^*. The probability of its collision with a value from a hash query in the same execution of F is at most $(q_{\text{hash}} + 1)/|Z_n^*|$ thus, the probability (taken over only the random choices of A) of A's failure to pretend is at most $q_{\text{sig}}(q_{\text{hash}} + 1)/|Z_n^*| \leq q_{\text{sig}}(q_{\text{hash}} + 1)2^{2-k}$ (because $|Z_n^*| = 4q_1q_2 > 2^{k-2}$). This is exactly the amount by which F's probability of success is reduced because of interaction with A rather than the real signer. Let $\delta = \varepsilon - q_{\text{sig}}(q_{\text{hash}} + 1)2^{2-k}$.

Let ε_b be the probability that F produces a successful forgery and that its break-in query occurs in time period b. Clearly, $\delta = \sum_{b=2}^{T+1} \varepsilon_b$ (if $b = 1$, then F cannot forge for any time period). Assume now that A picked $j = b - 1$ for some fixed b. The probability of that is $1/T$.

We will now calculate the probability of the event that F outputs a valid forgery based on the same hash query both times and that the hash query was answered differently the second time and that the break-in query was b both times. Let $p_{h,b}$ be the probability that, in one run, F produces a valid forgery based on hash query number h after break-in query in time period b. Clearly,

$$\varepsilon_b = \sum_{h=1}^{q_{\text{hash}}+1} p_{h,b}$$

Let $p_{h,b,S}$ (for a sufficiently long binary string S of length m) be the probability that, in one run, F produces a valid forgery based on hash query number h after break-in query in time period b, given that the string S was used to determine the random tape of F and the responses to all the oracle queries of F until (and not including) the h-th hash query. We have that

$$2^m p_{h,b} = \sum_{S \in \{0,1\}^m} p_{h,b,S}.$$

Given such a fixed string S, the probability that F produces a valid forgery based on the hash query number h after break-in query in time period b in both runs is $p_{h,b,S}^2$ (because the first forgery is now independent of the second forgery). The additional requirement that the answer to the hash query in the second run be different reduces this probability to $p_{h,b,S}(p_{h,b,S} - 2^{-l})$. Thus, the probability $q_{h,b}$ that F produces a valid forgery based on the hash query number h in both runs and that the answer to the hash query is different in the second run and that the break-in query was b in both runs is

$$\begin{aligned} q_{h,b} &= \sum_{S \in \{0,1\}^m} 2^{-m} p_{h,b,S}(p_{h,b,S} - 2^{-l}) \\ &= 2^{-m} \left(\sum_{S \in \{0,1\}^m} p_{h,b,S}^2 - 2^{-l} \sum_{S \in \{0,1\}^m} p_{h,b,S} \right) \\ &\geq \frac{2^{-m} \left(p_{h,b} 2^m\right)^2}{2^m} - 2^{-l} p_{h,b} = p_{h,b}^2 - 2^{-l} p_{h,b} \end{aligned}$$

(by Lemma 2).

The probability that F outputs a valid forgery based on the same hash query both times and that the hash query was answered differently in the second run and that the break-in query occurred in time period i is now

$$\sum_{h=1}^{q_{\text{hash}}+1} q_{h,b} \geq \sum_{h=1}^{q_{\text{hash}}+1} p_{h,b}^2 - \sum_{h=1}^{q_{\text{hash}}+1} 2^{-l} p_{h,b} \geq \frac{\varepsilon_b^2}{q_{\text{hash}}+1} - 2^{-l}\varepsilon_b$$

(by Lemma 2).

Note that if this happens, then the forgery occurs in time period $i < b = j+1$ (because the forgery has to occur before the break-in query), so A will be able to take a root of α.

Finally, we again use Lemma 2 to remove the assumption that A picked $j = b-1$ as the time period to get the probability of A's success:

$$\varepsilon' \geq \frac{1}{T} \sum_{i=2}^{T+1} \left(\frac{\varepsilon_b^2}{q_{\text{hash}}+1} - 2^{-l}\varepsilon_b \right) \geq \frac{\delta^2}{T^2(q_{\text{hash}}+1)} - \frac{\delta}{2^l T}. \qquad \square$$