Full version of an extended abstract published in Proceedings of ACM CCS 2006, ACM Press, 2006. Available from the IACR Cryptology ePrint Archive as Report 2006/297.

Forward-Secure Signatures with Untrusted Update

Xavier Boyen * Hovav Shacham †‡ Emily Shen § Brent Waters ¶‖

September 2, 2006

Abstract

In most forward-secure signature constructions, a program that updates a user's private signing key must have full access to the private key. Unfortunately, these schemes are incompatible with several security architectures including Gnu Privacy Guard (GPG) and S/MIME, where the private key is encrypted under a user password as a "second factor" of security, in case the private key storage is corrupted, but the password is not.

We introduce the concept of forward-secure signatures with untrusted update, where the key update can be performed on an encrypted version of the key. Forward secure signatures with untrusted update allow us to add forward security to signatures, while still keeping passwords as a second factor of security. We provide a construction that has performance characteristics comparable with the best existing forward-secure signatures. In addition, we describe how to modify the Bellare-Miner forward secure signature scheme to achieve untrusted update.

1 Introduction

One problem commonly faced in security research is how to limit damage when an attacker compromises a system and private secrets are exposed. Ross Anderson, in an invited talk, originally proposed a signature scheme known as Forward-Secure Signatures [3] that was meant to mitigate the damage when private signature keys were exposed. In his proposal, each signature would be associated with the current time period in addition to the signed message. After each time interval, a user's private signing key is updated such that it can no longer be used to sign for past time periods. In this manner, if a user's private key is compromised at a given time period, the attacker is unable to forge signatures that appear to come from any earlier time period.

Anderson's original solution was quite simple. At setup time a user would issue himself a separate certificate and private key for every time period. As each time period passed, the private key for that period would be deleted. While the solution was quite elegant, it had the drawback that the private key size of the user grew linearly with the number of time periods, making it

*Voltage Security Inc. — `xb@boyen.org`

†Weizmann Institute of Science — `hovav.shacham@weizmann.ac.il`

‡Supported by a Koshland Scholars Program fellowship.

§Stanford University — `emily@cs.stanford.edu`

¶SRI International — `bwaters@csl.sri.com`

‖Supported by NFS and DHS. This material is based upon work supported by the Department of Homeland Security (DHS) and the Department of Interior (DOI) under Contract No. NBCHF040146. Any opinions, finding and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DHS and DOI.

rather inefficient and infeasible for systems requiring a small storage space (such as smartcards). Bellare and Miner [4] formalized the notion of Forward-Secure Signatures and provided two schemes with improved private storage requirements. Several constructions followed [25, 22, 19, 21, 1, 2] that provided different tradeoffs in the metrics of private key storage, signature size, and the computational time required for setup, key update, signing, and verifying.

While there has been considerable (and successful) effort put into achieving constructions with sufficient efficiency for practical use, in order for a new primitive to be adopted it must integrate into existing software architectures. In addition, it should be as transparent as possible—the added burden on the user should be minimal. One feature of several cryptographic software suites, including the popular Pretty Good Privacy suite [34], is that a user's private key is encrypted under some additional secret, typically a password. The added value of encrypting the private key is that it provides a "second factor" for security in case the storage is compromised.

Unfortunately, most existing forward-secure signature constructions are difficult to integrate into this model. The primary difficulty is that the "update" algorithms need to access the private key unencrypted in order to move it forward. Given this limitation, the software will either need to require that the user intervene each time it updates the key or it will need to forgo the second factor altogether. The former option is undesirable for several reasons: first, it places a burden on the users to take an action at regular intervals to update their keys, something they will most likely come to resent; second, it makes some update schedules (say, every hour) infeasible; third, since updates are not automatic, keys are likely to be often not fully updated, and, if an non-updated key is compromised, the attacker can produce signatures for past time periods—precisely the threat that forward-secure signatures are supposed to prevent. The second choice is also unfavorable since it is unclear that achieving forward security is worth the tradeoff of abandoning the extra security provided by keeping the keys encrypted under the second factor; developers of security software will be unlikely to adopt a new feature in favor of dropping an old one.

Our Contribution. We introduce *forward-secure signatures with untrusted update*, a signature primitive that allows a program to update, i.e., move forward in time, an "encrypted" version of the private key. To sign a message an algorithm must have an additional secret key, the second factor, which in practice is a password provided by the user. The required security properties are both that an attacker with access to just the encrypted key cannot forge signatures and that the scheme will maintain the traditional forward-secure properties.

We observe that the original proposal of Anderson can actually be easily modified to achieve these properties—simply encrypt each of the private keys under the secret key. However, as discussed above, this solution requires an unreasonable amount of private key storage. Our goal is to design a forward secure scheme with comparable or better performance and security properties than existing schemes.

We create a (non-trivial) forward-secure signature scheme with untrusted updates. Our construction is "tree-based" and the underlying structure is similar to recent hierarchical identity-based encryption schemes [8, 6] where the private keys are composed of elements from bilinear groups. Using this structure, we are able to encrypt private keys in such a way that a third party can perform a homomorphic key update operation on them.

Our scheme is quite efficient with signatures consisting of three group elements, private keys of $O(\log(n)^2)$ group elements, and constant encryption, verification, update, and setup times. In addition, our scheme is provably secure without random oracles. These features are actually interesting in their own right, and we regard the forward-secure construction as being of interest even if we disregard the untrusted update property.

While our primary contribution is the introduction of a new forward-secure signature scheme with untrusted update, one might desire to realize untrusted update in a different signature scheme that has different security or performance tradeoffs. We show that the concept of untrusted update can be applied in other schemes by sketching how to modify the Bellare-Miner [4] forward-secure signature scheme to achieve untrusted update.

Finally, to demonstrate the practical applicability of our scheme, we provide an implementation of our construction. We create an API for our construction.

1.1 Organization

We begin by discussing related work in Section 2. Next, we give a formal description of a forward-secure signature scheme with untrusted update and definitons of security in Section 3. We present some background information on bilinear groups and our assumptions in Section 4. Then we present our construction and a proof of its security in Section 5. Additionally, we describe in Section 6 how to equip the Bellare-Miner [4] forward-secure signature with an untrusted update. We follow by describing a software implementation of our scheme in Section 7. Finally, we conclude in Section 8.

2 Related Work

Originally, forward security was introduced for key exchange protocols [16]. Anderson's original suggestion was for the user to store a separate private key for each time period. Bellare and Miner [4] later formalized the notion of forward-secure signatures.

Following work can roughly be divided into two classes. The first comprises generic constructions that need not necessarily require random oracles. The first of these is the tree construction of Bellare and Miner [4]. In this construction a generic signature scheme is used to build a binary tree from chains of certificates where leaves correspond to time periods. The private key storage, signature time, and verification time will all be a multiplicative factor of $O(\log(T))$ longer than the original signature scheme, where T is the number of time periods. Malkin, Micciancio, and Miner [25] apply Merkle trees [26] so that signing and verifying requires $O(\log(n))$ hashes (instead of signing or verifying operations). This comes at some additional expense during setup and key generation, though. They additionally show a method for combining various tree-based schemes in order to make various tradeoffs. Cronin et al. [9] provide an evaluation of the practical performance of these schemes and create an open-source forward-secure signature library. Krawczyk [22] presents a generic method for keeping a short private secret key on trusted storage; however, one must still maintain some (possibly untrusted) storage linear in T for the signer.

The other class of forward-secure signatures comprises specific random oracle-based schemes. The first of these was due to Bellare and Miner [4] in which they achieve short signatures with fast key update by applying Ong-Schnorr signatures [28]; however, the verification procedure is linear in T. Abdalla and Reyzin [2] later show tradeoffs in the computational time with signature and public key size. Itkis and Reyzin [19] propose a scheme with highly efficient signature and verification times based on Guillou-Quisquater signatures [15]. Although their basic technique requires an expensive update, they show how to apply certain pebbling techniques to achieve constant update time while storing just $O(\log(T))$ elements. Finally, Kozlov and Reyzin [21] give a scheme with fast key update.

In addition, there has been work on related subjects such as key-insulated update and intrusion resilient signature schemes [10, 11, 20, 18] and applications of forward security to group signatures [32] and threshold cryptography [1].

Canetti, Halevi and Katz [8] show how Hierarchical Identity-Based Encryption [14, 17] (HIBE) can be used to achieve forward-secure encryption. Boneh, Boyen, and Goh [6] later show how certain HIBE ciphertexts can be compressed to a constant size. In our work we apply Naor's observation (stated in [7]) that private keys from an Identity-Based Encryption [31] system can be viewed as signatures on a given identity. We use a particular version of hierarchical signatures to achieve forward security with short signatures without random oracles.

3 Definitions

We describe forward-secure signature schemes and give a formal definition for their security. A forward-secure signature scheme is made up of four algorithms:

KeyGen(T)**:** The setup algorithm takes in an integer, T, the number of time periods and outputs a public verification key VK, the encrypted signing key EncSK, and another second factor secret decryption key DecK. The current time period identifier, ID, is initially set to 1. The time period is embedded within the encrypted signing key.

Update$(\mathsf{EncSK}, \mathsf{ID}')$**:** The update algorithm takes in the encrypted signing key at some time period ID and outputs a new encrypted signing key EncSK' for time period $\mathsf{ID}' > \mathsf{ID}$. After this the previous signing key is erased. If $\mathsf{ID}' \geq T$, then the old key is just erased and there is no new key. This algorithm does not require the decryption key.

Sign$(\mathsf{EncSK}, \mathsf{DecK}, M)$**:** The signing algorithm takes as input the encrypted signing key EncSK, the second factor decryption key DecK, and a message M. It outputs a signature S for the time period ID that is embedded in the signing key. The time period may be included as part of the signature.

Verify(S, M, VK)**:** The verification algorithm takes as input a signer's verification key VK, a message M, and a signature S. It outputs either `valid` or `invalid`.

3.1 Security Model

We now define the security of forward-secure signatures with untrusted update in terms of two games.

3.1.1 Forward Security

The first security game captures the "traditional" notions of existential unforgeability and forward security. The game is played between an adversary $\mathcal{A}$ and a challenger $\mathcal{B}$, and proceeds in three phases.

Key Generation. The challenger runs the Setup algorithm and gives the adversary the verification key VK, and the second factor decryption key DecK. The time period t is set to 1.

Interactive Queries. In the query phase the adversary can issue three types of requests in an adaptive, interactive manner:

Sign: The adversary can query the challenger to sign a message M on the current time period ID; the challenger will then return a signature S.

Update: The adversary can request that the challenger execute the update algorithm, in which case the time period will be increased to a new value ID' chosen by the adversary.

Corrupt: The adversary can request that the challenger hand out all its keys at the current time period. The challenger returns the encrypted signing key EncSK for the current time period.

The adversary can repeatedly make Sign and Update queries; however, once he makes a Corrupt query the game moves to the next phase.

Final Forgery. Let ID' be the time period at which the Corrupt query was issued. The adversary produces a forgery, consisting of a time, message, signature tuple $(\mathsf{ID}^*, M^*, S^*)$. The adversary is successful if $\mathsf{ID}^* < \mathsf{ID}'$, the signature verifies for time ID^*, and the adversary had not queried for a signature on M^* at the exact time period ID^*.

We let $\mathsf{AdvFS}_{\mathcal{A}}$ denote the advantage of an algorithm $\mathcal{A}$ in the forward-security game.

3.1.2 Update Security

This game captures the notion of security against an adversary that controls the storage of the encrypted signing key, but not the second factor decryption key.

Key Generation. The challenger runs the Setup algorithm and gives the adversary the verification key, VK, and the initial encrypted signing key, EncSK. The time period ID is set to 1.

Query Phase. The adversary can issue two types of interactive requests:

Sign: The adversary can query the challenger to sign a message M under a key EncSK' specified by the adversary, for the current time period ID. The challenger *must* output a signature S if the given key EncSK' appears well formed (for the current time period ID). It *may* return an error symbol $\perp$ whenever it can demonstrate that EncSK' is not well formed.

Update: The adversary can request that the challenger update the clock to a new time $\mathsf{ID}' > \mathsf{ID}$, of the adversary's choice.

The adversary can repeatedly make Sign and Update queries, and at some point will choose to move to the next phase.

Final Forgery. At last, the adversary produces a time, message, and signature tuple $(\mathsf{ID}^*, M^*, S^*)$. The adversary is successful if the signature verifies for time ID^*, and the adversary had not queried for a signature on M^* at time period ID^*.

We let $\mathsf{AdvUS}_{\mathcal{A}}$ denote the advantage of an algorithm $\mathcal{A}$ in the update security game.

The "encrypted" signing key EncSK need not be encrypted in the traditional semantically secure sense. The only requirement is that the key be blinded or rendered inoperative in such a way that no adversary can gain a non-negligible advantage in the update security game. In other words, the encrypted signing key EncSK should be useless by itself to produce signatures.

The decryption key DecK is viewed in this model as being output by the key generation algorithm. In Section 7 we will address the issue of letting DecK be derived from a user password, as will often be done in practice.

4 Background

Before describing our scheme, we briefly review a few notions.

4.1 Bilinear Groups and Pairings

We review the usual notions of bilinear groups and bilinear maps defined over them [13, 29]. We use a multiplicative notation for the group operations. For simplicity, we restrict our attention to "symmetric" bilinear maps, while noting that our constructions can be advantageously generalized to use asymmetric pairings.

Let $\mathbb{G}$ and $\mathbb{G}_t$ be two cyclic groups of prime order p, and let g be a generator of $\mathbb{G}$. A symmetric bilinear map over $\mathbb{G}$ is a non-constant function $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$ such that $e(u^a, v^b) = e(u, v)^{ab}$ for all $u, v \in \mathbb{G}$ and all $a, b \in \mathbb{Z}$. We say that an (infinite) family of groups $\mathbb{G}$ with these properties forms a bilinear group family if the group operation and the bilinear map admit $O(\text{poly} \log |p|)$-time algorithms. It is common to refer to $\log |p|$, or an appropriately rounded multiple thereof, as the security parameter.

4.2 Computational Complexity Assumptions

Many complexity assumptions have been proposed in the context of bilinear pairings. In this paper, we make use of the Computational Diffie-Hellman assumption in bilinear groups (CDH), and the Bilinear Diffie-Hellman Inversion assumption (BDHI).

The CDH assumption in a bilinear group $\mathbb{G}$ is very similar to the familiar CDH assumption: given group elements $g, g^a, g^b \in \mathbb{G}$, it assumes that it is infeasible to compute $g^{ab} \in \mathbb{G}$. One important distinction, however, is that in a bilinear group the corresponding DDH problem is easy: given $g, g^a, g^b, Z \in \mathbb{G}$ we can tell whether $Z = g^{ab}$ by testing whether the equality $e(g^a, g^b) = e(g, Z)$ holds in $\mathbb{G}_t$. The CDH assumption in a bilinear group thus makes a stronger statement than it does classically.

The BDHI assumption in a bilinear group $\mathbb{G}$ originates from [27, 5]. Given a parameter $\ell \geq 1$, the ℓ-BDHI assumption in $\mathbb{G}$ states the following: given $g, g^\alpha, g^{\alpha^2}, \ldots, g^{\alpha^\ell} \in \mathbb{G}$, it is infeasible to compute $e(g, g)^{1/\alpha} \in \mathbb{G}_t$.

Definition 4.1. We say that the ℓ-BDHI assumption holds in $\mathbb{G}$ if no efficient algorithm can solve a random instance with non-negligible probability. For $i = 1, \ldots, \ell$, let $g_i = g^{(\alpha^i)} \in \mathbb{G}$. An algorithm $\mathcal{A}$ has an advantage ϵ in solving the ℓ-BDHI problem in $\mathbb{G}$ if

$$\Pr\left[\mathcal{A}(g, g_1, \ldots, g_\ell) = e(g,g)^{(\alpha^{-1})}\right] \geq \epsilon \ .$$

The probability is over the random choice of $g \in \mathbb{G}$ and α in $\mathbb{Z}_p$, and the random bits used by $\mathcal{A}$. We then say that the computational (t, ϵ, ℓ)-BDHI assumption holds in $\mathbb{G}$ if no t-time algorithm has advantage at least ϵ in solving a random instance of the ℓ-BDHI problem in $\mathbb{G}$.

Clearly, for all ℓ, the $(\ell + 1)$-BDHI assumption is at least as strong as the ℓ-BDHI assumption. In addition, the 1-BDHI assumption is itself at least as strong as the CDH assumption in $\mathbb{G}$. We shall still mention and use the CDH assumption in $\mathbb{G}$ as it makes certain proofs clearer.

5 Construction

It is instructive to first understand the intuition behind the forward-secure signature scheme without the untrusted update property. The scheme is roughly based on a hierarchical identity-based encryption (HIBE) structure [17, 14]. We use a similar approach to Canetti, Halevi, and Katz [8] in that we use a binary tree hierarchy to represent a discrete notion of time: we map the leaves of the tree to the corresponding time periods. A secret key holder will store the private keys material for a set of at most ℓ nodes, from which the private keys for the current and all future time periods can be derived, where ℓ is the height or depth of the binary tree.

To sign a message, the signer will use the private key of the current time period. By applying the compression techniques of Boneh, Boyen, and Goh [6], this private key can be expressed in two group elements, even though it is for an ℓ-level identity. The signature is obtained by appending to the hierarchy a final "identity" that is dependent on the message; we use the Waters hash [33] at this level to obtain existential unforgeability.

By combining all of these ideas we can obtain a forward-secure scheme with constant size signatures that is provably secure without random oracles. To obtain the untrusted update property, at key creation time we simply multiply the initial private keys by a second factor "decryption key" DecK, which we will assume to be a secret group element. (In practice, one can hash a secret bitstring to obtain the group element.) Since the private keys are blinded by DecK, an attacker with access to the private storage will not be able to sign messages. However, the user can divide out DecK and recover the true private keys of the original scheme outlined above in order to sign messages. Finally, we observe that the update procedure of our particular HIBE-based signature scheme will produce private key nodes that are still a factor of DecK away from a "real" private key. The fact that this remains consistent during an HIBE *Derive* (i.e., key delegation) procedure is what allows an untrusted entity perform an update.

We now give a detailed construction of our scheme and then state our formal theorems of security.

5.1 Scheme

Let $\mathbb{G}$ be a bilinear group of prime order p, and let $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$ be a symmetric bilinear map.

Messages to be signed are taken as fixed-length binary strings of m bits, for simplicity. Furthermore, for certain proofs (such as of the Update Security property), it is convenient to forbid that signatures on the same message be ever issued at two different time periods. One simple way to enforce this restriction is to embed the time period within the m bits to be signed, for example, as $M = \text{Time}\|\text{Msg}$ (with the drawback of reducing the available space for the message itself). Another way is to hash the message together with the time using a collision-resistant hash function $H : \{0,1\}^* \to \{0,1\}^m$, and sign the result $M = H(\text{Time}\|\text{Msg})$. Yet another way involves a universal one-way hash family $H : \{0,1\}^{m'} \times \{0,1\}^* \to \{0,1\}^{m''}$ such that $m' + m'' = m$; the signer would pick a random index $R \in \{0,1\}^{m'}$ into the UOWHF family, and then sign the concatenation $M = R\|H(R, \text{Time}\|\text{Msg})$.

Using a hash function such as these also has the advantage of letting us sign arbitrarily long messages. In the description that follows, we shall keep all this in mind though we treat M as an m-bit fixed-length binary string.

Time is modeled discretely as a sequence of 2^ℓ atomic time periods, arranged as the leaves of a binary tree of depth ℓ, in chronological order. A time period is identified by an ℓ-bit integer $\mathsf{ID} = (\mathrm{I}_1, \ldots, \mathrm{I}_\ell) \in \{0,1\}^\ell$ (though for convenience we exclude the zero-th period 0^ℓ and let the first period be represented by $0^{\ell-1}1$). The bits I_1 to I_ℓ are ordered from the top to the bottom of the tree, while a 0 and a 1 respectively indicate the first and second branch in the order of traversal. It follows that the traversal of the leaves, the chronology of the time periods, and the numerical values of their identifiers, all obey the same ordering.

For $j = 1, \ldots, \ell + 1$, we define a time period's "second sibling at depth j", as

$$\mathsf{sibling}(j, \mathsf{ID}) = \begin{cases} (\mathrm{I}_1, \ldots, \mathrm{I}_{j-1}, 1) & \text{if } j < \ell + 1 \text{ and } \mathrm{I}_j = 0 \ , \\ \perp & \text{if } j < \ell + 1 \text{ and } \mathrm{I}_j = 1 \ , \\ \mathsf{ID} & \text{if } j = \ell + 1 \ . \end{cases}$$

A second sibling at depth j is either $\perp$, or a j-bit string that is never a prefix of ID — except that for notational convenience we pose $\mathsf{sibling}(\ell + 1, \mathsf{ID}) = \mathsf{ID}$ at the fictitious depth $\ell + 1$.

Last, we let $\mathsf{bit}(i, S)$ denote the i-th bit from the string $S \in \{0,1\}^*$, that is, for $1 \le i \le n$ and $S = (s_1, s_2 \ldots, s_n) \in \{0,1\}^n$, we have $\mathsf{bit}(i, S) = s_i \in \{0,1\}$.

Our forward-secure signature scheme with untrusted updates works as follows:

KeyGen(ℓ, m)**:** Let 2^ℓ be the number of time periods, and $\{0,1\}^m$ the message space. The generation of a random initial set of keys proceeds as follows. Select two random integers $\nu, \omega \in \mathbb{Z}_p$, and a few random group elements $g, h_0, h_1, \ldots, h_\ell, f_0, f_1, \ldots, f_m \in \mathbb{G}$. We fix $\mathsf{ID} = 0^{\ell-1}1$.

For each $j = 1, \ldots, \ell + 1$, we let $k_j = \mathsf{sibling}(j, \mathsf{ID})$. Specifically, $(k_1,\ k_2,\ \ldots,\ k_\ell,\ k_{\ell+1}) = (1,\ 01,\ \ldots,\ 0^{\ell-2}1,\ \perp,\ 0^{\ell-1}1)$. For each k_j, a private key component K_j is computed as follows. If $k_j = \perp$, then $K_j = \perp$. Otherwise, pick a random integer $r_j \in \mathbb{Z}_p$, and let

$$K_j = \left(g^{\nu+\omega} \cdot \Big(h_0 \cdot \underbrace{h_1^{\mathsf{bit}(1,k_j)} \cdot \ldots \cdot h_j^{\mathsf{bit}(j,k_j)}}_{|k_j| \text{ factors}} \Big)^{r_j},\ \ g^{r_j},\ \ \underbrace{h_{j+1}^{r_j},\ \ \ldots,\ \ h_\ell^{r_j}}_{\ell - |k_j| \text{ components}} \right) .$$

Since for $j < \ell$ we have $k_j = 0^{j-1}1$, we obtain $K_j = \big(g^{\nu+\omega} \cdot (h_0 \cdot h_j)^{r_j},\ \ g^{r_j},\ \ h_{j+1}^{r_j},\ \ \ldots,\ \ h_\ell^{r_j}\big)$. For $j = \ell$, we get $K_\ell = \perp$. For $j = \ell + 1$, we end up with $K_{\ell+1} = \big(g^{\nu+\omega} \cdot (h_0 \cdot h_\ell)^{r_{\ell+1}},\ \ g^{r_{\ell+1}}\big)$ by letting $h_{\ell+1}^0 = 1$ in the general expression above even though $h_{\ell+1}$ is not defined.

The encrypted signing key for period $\mathsf{ID} = 0^{\ell-1}1$ and the "second factor" decryption key are

$$\mathsf{EncSK}_{0^{\ell-1}1} = \Big(\mathsf{ID},\ \ K_1,\ \ K_2,\ \ \ldots,\ \ K_{\ell-1},\ \ K_\ell = \perp,\ \ K_{\ell+1} \Big) , \qquad \mathsf{DecK} = g^{-\omega} \ .$$

Also compute $V = e(g,g)^{\nu}$ and $W = e(g,g)^{\omega}$. The public verification key is given by

$$\mathsf{VK} = \Big(g,\ \ V,\ \ W,\ \ h_0,\ \ h_1,\ \ \ldots,\ \ h_\ell,\ \ f_0,\ \ f_1,\ \ \ldots,\ \ f_m\Big)\ .$$

CheckKey$(\mathsf{EncSK}_{\mathsf{ID}}, \mathsf{VK})$**:** To verify that an encrypted key $\mathsf{EncSK}_{\mathsf{ID}}$ is valid for a public key VK, proceed as follows.

Parse $\mathsf{EncSK}_{\mathsf{ID}}$ as $(\mathsf{ID}, K_1, \ldots, K_\ell, K_{\ell+1})$. For $j = 1, \ldots, \ell+1$, let $k_j = \mathsf{sibling}(j, \mathsf{ID})$, and check that $k_j = \perp$ if and only if $K_j = \perp$. Then, for each j such that $k_j \neq \perp$, parse K_j as $(a_0, a_1, b_{j+1}, \ldots, b_\ell)$, verify that $1 = V \cdot W \cdot e(a_0, g^{-1}) \cdot e\big(a_1, h_0 \prod_{i=1}^{j} h_i^{\mathsf{bit}(i,k_j)}\big)$ in $\mathbb{G}_t$ and that $\forall i = j+1, \ldots, \ell : e(a_1, h_i) = e(b_i, g)$ in $\mathbb{G}_t$. If all equalities are verified, output `valid`, otherwise output `invalid`.

Update$(\mathsf{EncSK}_{\mathsf{ID}}, \mathsf{ID}', \mathsf{VK})$**:** To update an encrypted signing key $\mathsf{EncSK}_{\mathsf{ID}}$ from time period ID to time period ID', given the verification key VK (but not the decryption key), proceed as follows. To start, parse $\mathsf{EncSK}_{\mathsf{ID}}$ as $(\mathsf{ID}, K_1, \ldots, K_\ell, K_{\ell+1})$, and ascertain that $0^\ell < \mathsf{ID} \leq \mathsf{ID}' \in \{0,1\}^\ell$.

Let j, j' denote indices in the range $1, \ldots, \ell+1$. Let $k_j = \mathsf{sibling}(j, \mathsf{ID})$ and $k'_{j'} = \mathsf{sibling}(j', \mathsf{ID}')$. By construction, each non-$\perp$ string $k'_{j'}$ contains exactly one of the strings k_j as a prefix. Formally: $\forall j' \in \{1, \ldots, \ell+1\}$, either $k'_{j'} = \perp$, or $\exists! j \in \{1, \ldots, j'\}$ s.t. $k'_{j'} = k_j \| s$ for some string $s \in \{0,1\}^{|k'_{j'}| - |k_j|}$.

For all $j' = 1, \ldots, \ell+1$, we construct the j'-th component $K'_{j'}$ of the updated key as follows. If $k'_{j'} = \perp$, then set $K'_{j'} = \perp$. If $k'_{j'} = k_j$, then set $K'_{j'} = K_j$. Otherwise, determine the index $j < j'$ such that $k'_{j'} = k_j \| s$ for some suffix string s, parse the j-th component of $\mathsf{EncSK}_{\mathsf{ID}}$ as $K_j = (a_0, a_1, b_{j+1}, \ldots, b_\ell)$, pick a fresh random $r_{j'} \in \mathbb{Z}_p$, and set

$$\begin{aligned} K'_{j'} &= (a'_0, a'_1, b'_{j'+1}, \ldots, b'_\ell) \\ &= \Bigg(a_0 \cdot \Big(\underbrace{b_{j+1}^{\mathsf{bit}(j+1,k'_{j'})} \cdot \ldots \cdot b_{j'}^{\mathsf{bit}(j',k'_{j'})}}_{|k'_{j'}| - |k_j| \text{ factors}} \Big) \cdot \Big(h_0 \cdot \underbrace{h_1^{\mathsf{bit}(1,k'_{j'})} \cdot \ldots \cdot h_{j'}^{\mathsf{bit}(j',k'_{j'})}}_{|k'_{j'}| \text{ factors}} \Big)^{r_{j'}},\ \ a_1 \cdot g^{r_{j'}}, \\ &\qquad b_{j'+1} \cdot h_{j'+1}^{r_{j'}},\ \ \ldots,\ \ b_\ell \cdot h_\ell^{r_{j'}} \Bigg)\ . \end{aligned}$$

Once all the $K'_{j'}$ have been computed, output the new encrypted key for period ID' as

$$\mathsf{EncSK}_{\mathsf{ID}'} = \big(\mathsf{ID}',\ \ K'_1,\ \ \ldots,\ \ K'_{\ell+1}\big)\ .$$

Sign$(\mathsf{EncSK}_{\mathsf{ID}}, \mathsf{DecK}, \mathsf{VK})$ **:** To sign a message $M \in \{0,1\}^m$ using the encrypted signing key $\mathsf{EncSK}_{\mathsf{ID}}$ and the decryption key DecK, proceed as follows.

Parse $\mathsf{EncSK}_{\mathsf{ID}} = (\mathsf{ID}, K_1, \ldots, K_\ell, K_{\ell+1})$, and then parse $K_{\ell+1} = (a_0, a_1) \neq \perp$. The elements a_0 and a_1 and the time period ID are all that we need from $\mathsf{EncSK}_{\mathsf{ID}}$. At this point, if the key is not trusted (e.g., being the result of an untrusted update), the signer needs to ensure that $1 = V \cdot W \cdot e(a_0, g^{-1}) \cdot e\big(a_1, h_0 \prod_{i=1}^{\ell} h_i^{\mathsf{bit}(i,\mathsf{ID})}\big)$ in $\mathbb{G}_t$. If this test fails, output $\perp$ and halt.

Otherwise, to produce a signature, pick two random integers $r, s \in \mathbb{Z}_p$, and output

$$S_{\mathsf{ID}}(M) = \Bigg(\mathsf{ID},\ \ \mathsf{DecK} \cdot a_0 \cdot \Big(h_0 \cdot \prod_{i=1}^{\ell} h_i^{\mathsf{bit}(i,\mathsf{ID})} \Big)^{r} \cdot \Big(f_0 \cdot \prod_{j=1}^{m} f_j^{\mathsf{bit}(j,M)} \Big)^{s},\ \ a_1 \cdot g^r,\ \ g^s \Bigg)\ .$$

Verify(S, M, VK)**:** To verify a signature $S = (\mathsf{ID}, s_0, s_1, s_2)$ on a message $M \in \{0,1\}^m$ with respect to a verification key VK, it suffices to check the following equality in $\mathbb{G}_t$:

$$1 = V \cdot e\big(s_0, g^{-1}\big) \cdot e\Big(s_1, h_0 \prod_{i=1}^{\ell} h_i^{\mathsf{bit}(i,\mathsf{ID})}\Big) \cdot e\Big(s_2, f_0 \prod_{j=1}^{m} f_j^{\mathsf{bit}(j,M)}\Big) \ .$$

Output `valid` if the equality holds, and `invalid` if it does not.

Observe that for any time period, the signature contains only three group elements in addition to the time period identifier ID. Signature verification is also fairly fast as it requires only three pairings.

Validating Untrusted Keys. The test procedure *CheckKey* serves to completely validate an encrypted signing key $\mathsf{EncSK}_{\mathsf{ID}}$. Without this check, certain corruptions of $\mathsf{EncSK}_{\mathsf{ID}}$ may not be immediately apparent, but will creep up in the update process, and eventually surface as we try to sign at some future time ID'. The basic check in the *Sign* algorithm is sufficient to prevent a signature from being created incorrectly, and thus *CheckKey* is superfluous in the formal proof of security. In practice, however, *CheckKey* is a useful discretionary test that should be performed before overwriting an old key with an new key from an untrusted source.

We also note that, as written, *CheckKey* implicitly assumes that the checking algorithm has an uncorrupted version of the public key VK when validating EncSK, in addition to DecK. This is potentially problematic if an attacker controlling the key storage can corrupt VK as well as EncSK. However, in practice we can protect against this attack by having the key DecK include a MAC key in addition to the second factor decryption key. The MAC key will be used by *KeyGen* to append an authentication code to the public key VK, and later by *CheckKey* to check the integrity of VK against that code. For simplicity we stick to the original model in the formal proofs.

Re-randomization Issues. For performance reasons, the *Update* procedure does not fully re-randomize the encrypted key $\mathsf{EncSK}_{\mathsf{ID}}$ upon all invocations. In particular, running a "zero-step" update of the form *Update*$(\mathsf{EncSK}_{\mathsf{ID}}, \mathsf{ID}', \mathsf{VK})$ with $\mathsf{ID}' = \mathsf{ID}$ simply outputs the given key, unaltered. We could have designed *Update* to recompute each component $K'_{j'}$, even in the case where $k'_{j'} = k_j \neq \perp$, causing the key to be fully re-randomized no matter how small the update. However, the selective re-randomization strategy gives us amortized constant time for single-step updates, which the indiscriminate strategy does not.

In the *Sign* procedure, we re-randomize the two group elements from $\mathsf{EncSK}_{\mathsf{ID}}$ that intervene in the signing process, before each signature. This ensures that the signatures are jointly uniformly distributed (over the space of valid signatures for given times and messages), which helps us keep the security proofs reasonably simple.

Very Fine Time Granularities. Our update algorithm is general in the sense that it lets us jump to any time period in the future in a single operation, as opposed to the next period only. The possibility of making large jumps greatly simplifies the updating task in the case where many time periods have elapsed since the last update. In turn, this makes it possible to have an extremely fine-grained discretization of time (such as 1μs periods over a 10-year span), without significant performance degradation.

Achieving Constant Update Time. Another desirable feature to have is a true constant-time update when the time increment is 1, rather than the *amortized* constant-time we currently have. Canetti, Halevi, and Katz [8] show a tree-based method for achieving constant-time updates by associating interior nodes with time periods and doing an in-order traversal through the tree. We could apply the same method; however, we choose to use a simpler structure to better expose the novel features of our scheme. The CHK method also loses its advantage over the simpler approach under fine-grained discretizations of time—where updates are to cover many *micro-periods* at once—, which we expect to be more important in practice.

5.2 Security

We now state our two main security theorems. Detailed proofs may be found in the appendix.

Theorem 5.1. *Let $\mathcal{A}$ be an adversary that produces an existential forgery, in the forward security attack model, against the signature scheme instantiated for m-bit messages and T time periods. Assume that $\mathcal{A}$ makes no more than q queries, and succeeds with probability ϵ in time τ. Then there exists an algorithm $\mathcal{B}$ that solves the ℓ-BDHI problem in $\mathbb{G}$ in time $\tilde{\tau} \approx \tau$ with success probability $\tilde{\epsilon} \geq \epsilon/(4mqT)$.*

Proof. See Appendix A. □

Theorem 5.2. *Let $\mathcal{A}$ be an adversary that produces an existential forgery, in the update security attack model, against the forward secure signature scheme instantiated to accept m-bit messages. Assume that $\mathcal{A}$ makes no more than q queries, and succeeds with probability ϵ in time τ. Then there exists an algorithm $\mathcal{B}$ that solves the CDH problem in $\mathbb{G}$ in time $\tilde{\tau} \approx \tau$ with success probability $\tilde{\epsilon} \geq \epsilon/(4mq)$.*

Proof. See Appendix A. □

6 Adding Untrusted Update to the Bellare-Miner Scheme

In the previous section we achieved untrusted update from a new forward-secure signature scheme derived from HIBE. Our primary leverage for achieving untrusted update was the particular algebraic structure of the underlying scheme. This leads to the natural question of whether other existing number-theoretic forward-secure signature schemes have similar properties and can be modified to achieve untrusted update.

In this section we describe how to modify the Bellare-Miner [4] scheme to achieve untrusted update. We give short, intuitive descriptions of the schemes and proofs, and omit the details.

We briefly describe the number theoretic scheme of Bellare and Miner [4]. The key generation algorithm chooses a Blum-Williams integer $N = pq$ and publishes the public key as N and ℓ random points $U_1, \ldots, U_\ell \in \mathbb{Z}_N$, where ℓ depends on the security parameter of the scheme. If the scheme has T total time periods, the initial private keys for time period 1 will be $S_{1,1}, \ldots, S_{\ell,1} \in \mathbb{Z}_N$, where $S_{i,1}$ is the 2^{T+1}-th root of U_i. The factorization is discarded after key generation.

Key update is done simply by squaring each value of the secret key. The private keys for time period $j+1$ are computed as $S_{i,j+1} = S_{i,j}^2$ for all $1 \leq i \leq \ell$. To sign a message at a time period t, the signer proves non-interactively that he has the 2^{T+2-t}-th roots for all U_i using Fiat-Shamir [12] heuristic techniques in the random oracle model.

Adding untrusted update is rather straightforward. Let the second factor DecK be ℓ elements $\mathsf{DecK}_1, \ldots, \mathsf{DecK}_\ell$ of $\mathbb{Z}_N$. (In practice these can be generated by applying a hash function modeled as a random oracle to a shorter secret.) The new initial secret keys are constructed by multiplying in the blinding factors as $S'_{1,i} = S_{1,i}\,\mathsf{DecK}_i$ for all $1 \leq i \leq \ell$, where $S_{1,1}, \ldots, S_{\ell,1}$ are the initial secret keys from the original scheme. The key update algorithm is the same; it simply squares each component of the secret key. At time period t, the secret key components will be $S'_{t,i} = S_{t,i}\mathsf{DecK}^{2^{t-1}}$. To recover the private key component of the original scheme $S_{t,i}$, the signer simply needs to compute $\mathsf{DecK}^{2^{t-1}}$ and divide it out from $S'_{t,i}$; after this step the signer can sign the message as before. One drawback is that the signing algorithm will need to perform a number of squarings that is linear in ℓT. We can informally, argue that the scheme is secure against untrusted storage since the stored private keys will be just random group elements in $\mathbb{Z}_n$.

It would be interesting to see untrusted update added to other signature schemes. In particular the Itkis-Reyzin [19] scheme is of particular interest due to its desirable performance parameters. However, the straightforward methods to add untrusted update remove the performance benefits that made the scheme interesting to begin with. We leave the addition of untrusted update to other existing forward secure signature schemes (without significantly degrading performance) as future work.

7 Implementation and Applications

We present the implementation of our signature scheme. We first describe the core functionality and the interface to our implementation. Then we discuss our performance measurements. Finally, we describe some issues that arise in integrating our code with security application programs.

7.1 Software Implementation

We describe the API to our forward-secure signature functionality and report timing numbers. For the elliptic curve operations underlying our crypto code, we used Ben Lynn's PBC library [24]. PBC uses the GMP library [30] for its bignums and other math code. We expect to make the source for our library available under a GPL-compatible license.

7.1.1 Interface

We describe our API and explain some of the choices we made. In addition to the core functions we describe here, there are, of course, routines for such mundane operations as reading keys from and writing them to disk.

```
void fs_gen_sys_param(fs_sys_param_t param, pairing_t pairing);
void fs_gen(fs_public_key_t pk, fs_private_key_t sk, fs_dec_key_t dk,
            fs_sys_param_t param, unsigned int msg_len_bits,
            unsigned int num_intervals);
```

Key generation first requires system parameter generation. It is expected that applications will ship with preselected system parameters, so that users need not generate their own. The actual key generation algorithm outputs a public key `pk`, an encrypted secret key `sk`, and a second-factor `dk` that can be used to decrypt `sk` for the signing operation. We expect that applications will use a password obtained from the user to encrypt the second-factor `dk`. The algorithm takes several arguments: the system parameters generated earlier; the length of messages to be signed, typically 160 and the output of a collision-resistant hash function like SHA-1; and the number of intervals T over which the key can evolve.

```
int get_time_from_sk(fs_private_key_t sk);
int fs_check_key(fs_private_key_t sk, fs_public_key_t pk);
void fs_update(fs_private_key_t sk, fs_public_key_t pk);
```

Three functions are called for managing time periods. None of these requires the second factor `dk` to operate, so they can all be called without the user's involvement. The first, `get_time_from_sk`, returns the time period to which the (encrypted) private key has been updated. The second, `fs_check_key`, checks that the private key is indeed valid for signing and has not been tampered with. The idea is that the application uses this function as a self-test before requiring the user to enter a password so that she can sign a message. The third, `fs_update`, updates the key by a single time period. It can, obviously be called repeatedly to update the key to an arbitrary period.

```
void fs_sign(unsigned char *sig, unsigned char *msg, unsigned int msg_len_bits,
     fs_public_key_t pk, fs_private_key_t sk, fs_dec_key_t dk);
int fs_verify(unsigned char *sig, unsigned char *msg,
      unsigned int msg_len_bits, int time, fs_public_key_t pk);
```

Finally, two functions are provided for signing and verifying messages. The signing operation, `fs_sign`, transforms a message in `msg` to a signature in `sig`; the verification operation `fs_verify` checks that `sig` is a correct signature on `msg`. The signing operation requires the second-factor key `dk` along with the (encrypted) secret key `sk`; the verification algorithm obviously, requires only the public key `pk`.

The signing function generates a signature for the present time period, so it does not take as input the time period to use. An application should make sure, before signing, that it has updated the key to the correct time period. This check could be performed within `fs_sign` itself, but it could then be the case that the application and the library have different ideas of what the present time period should be.

Our verification function, on the other hand, takes a time period `time`. It is the application's job to determine what time period it expects the signature to be from. As with the signing algorithm, the intention is that all clock-to-time-period conversion be handled by a single location in the code to avoid what is (more literally here than usually) a time-of-check–time-of-use error. The algorithm required for this conversion is quite simple for most applications, doing simple arithmetic on the start and end times for the key's validity and the number T of time periods through which it evolves, along with the time at which the signature was generated. Some applications might impose additional constraints, however, so the code should be implemented in the application to ensure a consistent answer.

7.1.2 Timing

In Table 1, we present timing numbers for the basic operations we expose. We test our code for several choices of total time intervals: $T \in \{16, 32, 64, 128, 256\}$. The setup uses an MNT curve chosen for 1024-bit security. We sign 160-bit messages. To smooth out the timing, we ran the

Table 1: Times, in seconds, of forward-secure signing operations, for various total numbers of time intervals.

Operation	$T = 16$	$T = 32$	$T = 64$	$T = 128$	$T = 256$
Key generation	37.365	37.832	38.636	39.187	41.026
Signature generation	0.134	0.136	0.135	0.138	0.137
Signature verification	0.328	0.329	0.333	0.341	0.341
Key update (1 step)	0.109	0.146	0.168	0.193	0.202
Key validity check	0.205	0.205	0.208	0.213	0.214

Table 2: Times, in seconds, of forward-secure signing operations, for large total numbers of intervals. Times are averaged over the first 1024 time intervals.

Operation	$T = 2^{10}$	$T = 2^{20}$	$T = 2^{30}$
Key generation	41.99	51.23	66.98
Signature generation	0.14	0.14	0.14
Signature verification	0.34	0.33	0.33
Key update (1 step)	0.22	0.22	0.23
Key validity check	0.21	0.21	0.21

following procedure five times: generate key, then, for each of the T time periods, verify that the key is valid, sign and verify a message and update to the next period. Measurements were taken on a 2.8 GHz Pentium IV machine with 512 MB RAM, running OpenBSD 3.8.

In additon, in Table 2, we present timing numbers for our system when used with a large number of intervals: $T \in \{2^{10}, 2^{20}, 2^{30}\}$. To generate this table we followed the same procedure as above, but we ran the timing procedure only once and averaged the single-step key update time only through the first 1024 intervals.

All operations except key generation take less than one second to complete. We note that efficient pairing computation is an active research area, and new algorithms are likely to decrease the times we see.

7.2 Practical Considerations

We describe some details that must be considered when integrating our forward-secure signature code with an application.

7.2.1 Storing the Second Factor

In our implementation, the second factor key `dk` is generated uniformly at random in the course of the `fs_gen` function. It is expected that the application encrypt this second factor on disk by means of a user password. In our analysis of update security, we expected that `dk` is kept secret from the adversary. Thus if a password is used to encrypt dk, it is important that this password contain sufficient entropy to deter offline password guessing. The same caveat, of course, applies to any private key stored encrypted with a password. If a better source of entropy is available, the application can make use of it.

An alternative design strategy would have allowed the application to supply the second-factor key ω on its own, rather than having ω randomly chosen by `fs_gen`. The application could then generate ω using a user-supplied password. The resulting scheme would be somewhat more efficient. However, it would require tighter integration between our code's representation of bignums and the application's routines for extracting entropy from passwords, a requirement we deemed inadvisable.

7.2.2 Key Storage and Unattended Updates

The natural and correct choice for implementing automatic updates is for the application to set up a `cron` job on the user's behalf that updates the key at the appropriate interval. The cron job can check if a key update has been skipped (for example, because the system was powered down) and apply the updates necessary to bring the key to the current time period. An application making use of our library can also check whether updates have been missed when it is run by the user.

The security guarantees of our scheme mean that it is also possible for the encrypted private key to be stored on a remote server whose job it is to update it at every interval. The danger of this approach is that if the server retains old versions of the secret key then a compromise of the second factor will cause these old versions to be revealed, not just the version corresponding to the present time period.

7.2.3 Mapping Times to Time Periods

The security of a forward-secure signature scheme relies crucially on a proper mapping of signatures to the time period in which they were generated. For example, consider a backdated signature that claims, "I was made in time period j," but was in fact made in time period $j' > j$, and verifies only with the public key for period j'. An application that accepts this signature will violate the forward-security semantics, though no cryptographic flaw exists. As with much about signatures, the semantics of signature verification are what is important, but also what is hard to pin down; cf. Laurie and Bohm [23].

It is also important, but less so, that the signer correctly calculate the the appropriate time period with which to generate a signature. In the worst case, the resulting signature will not verify, since it will be deemed to have been backdated. The calculations are also easier for the signer, since it relies on the current time as reported by the OS (or a clock server), not the signing time listed in a maliciously generated signature. The secret signing key should be kept updated on disk, of course.

An upshot of this is that an application that verifies signatures from untrusted sources must be sure to make the user aware of the time in which it understands the signature to have been generated and, ideally, also the start and end of that time period. This is an extension of the backdating attack mentioned earlier. Even if a signature that claims to have been issued in time period j and was in fact issued then, the contents of the message might lead the user astray. For example, suppose document signing functionality with forward security is added to a word processing application. If the displays a checkmark next to the document to indicate that the signature was valid, an attacker who compromises Alice's signing key today can create a signature (correctly using the current time period) on a document that, in its body, prominently lists a date in the past. Now Bob, opening the document, will be tricked into believing that Alice generated it earlier, and might not discount it even if he hears from Alice that her key was recently compromised.

8 Conclusion

We introduced the notion of forward-secure signatures with untrusted update. With these signatures, private keys can be updated forward in time as they are kept in encrypted form, without first requiring decryption. This allows practical applications such as GPG to adopt the benefits of forward security, without forgoing the practice of encrypting the private keys under a second factor such as a user password.

We presented and proved secure a very efficient construction of forward-secure signatures with untrusted update, based on pairings. We also showed how to retrofit untrusted update into some existing forward-secure schemes, based on factoring. To validate the concept, we implemented our main (pairing-based) construction using the open-source GMP and PBC libraries, and obtained performance measurements.

References

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

[2] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In *ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, pages 116–129, London, UK, 2000. Springer-Verlag.

[3] Ross Anderson. Invited Lecture. 4th ACM Computer and Communications Security, 1997.

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

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

[6] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In *Advances in Cryptology—EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 440–456. Berlin: Springer-Verlag, 2005.

[7] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. Computing*, 32(3):586–615, 2003.

[8] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT*, pages 255–271, 2003.

[9] Eric Cronin, Sugih Jamin, Tal Malkin, and Patrick McDaniel. On the performance, feasibility, and use of forward-secure signatures. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications security*, pages 131–144, New York, NY, USA, 2003. ACM Press.

[10] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Key-insulated public key cryptosystems. In *EUROCRYPT*, pages 65–82, 2002.

[11] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Strong key-insulated signature schemes. In *Public Key Cryptography*, pages 130–144, 2003.

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

[13] Steven Galbraith. Pairings. In Ian F. Blake, Gadiel Seroussi, and Nigel Smart, editors, *Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society Lecture Notes*, chapter IX, pages 183–213. Cambridge University Press, 2005.

[14] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In *ASIACRYPT*, pages 548–566, 2002.

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

[16] Christoph G. Günther. An identity-based key-exchange protocol. In *EUROCRYPT*, pages 29–37, 1989.

[17] Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In *EUROCRYPT*, pages 466–481, 2002.

[18] Gene Itkis, Robert McNerney, and Scott Russell. Intrusion-resilient secure channels. In *ACNS*, pages 238–253, 2005.

[19] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 332–354, London, UK, 2001. Springer-Verlag.

[20] Gene Itkis and Leonid Reyzin. Sibir: Signer-base intrusion-resilient signatures. In *CRYPTO*, pages 499–514, 2002.

[21] Anton Kozlov and Leonid Reyzin. Forward-secure signatures with fast key update. In *SCN*, pages 241–256, 2002.

[22] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *CCS '00: Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 108–115, New York, NY, USA, 2000. ACM Press.

[23] Ben Laurie and Nicholas Bohm. Signatures: An interface between law and technology, January 2003. Online: `http://www.apache-ssl.org/tech-legal.pdf`.

[24] Ben Lynn. PBC library. Online: `http://rooster.stanford.edu/~ben/pbc/`.

[25] Tal Malkin, Daniele Micciancio, and Sara K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *EUROCRYPT*, pages 400–417, 2002.

[26] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[27] Shigeo Mitsunari, Ryuichi Sakai, and Masao Kasahara. A new traitor tracing. *IEICE Transactions on Fundamentals*, E850A(2):481–484, 2002.

[28] H. Ong and Claus-Peter Schnorr. Fast signature generation with a fiat shamir-like scheme. In *EUROCRYPT*, pages 432–440, 1990.

[29] Kenneth Paterson. Cryptography from pairings. In Ian F. Blake, Gadiel Seroussi, and Nigel Smart, editors, *Advances in Elliptic Curve Cryptography*, volume 317 of *London Mathematical Society Lecture Notes*, chapter X, pages 215–51. Cambridge University Press, 2005.

[30] GMP Project. The Gnu multiprecision arithmetic library. Online: `http://www.swox.com/gmp/`.

[31] Adi Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, pages 47–53, 1984.

[32] Dawn Xiaodong Song. Practical forward secure group signature schemes. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 225–234, New York, NY, USA, 2001. ACM Press.

[33] Brent Waters. Efficient identity-based encryption without random oracles. In *EUROCRYPT*, pages 114–127, 2005.

[34] Philip R. Zimmermann. *The Official PGP User's Guide*. The MIT Press, 1995. ISBN 0-262-74017-6.

A Proofs

We give detailed formal proofs of the security properties stated in Section 5.2.

For simplicity, and to get slightly stronger results, we base our security reductions on the so-called wBDHI* assumption [6], which is a slightly weaker and more convenient variant of the familiar BDHI assumption discussed earlier. It is immediate to show that for any $\ell \geq 1$, any algorithm that (τ, ϵ)-solves the ℓ-wBDHI* problem in a bilinear group $\mathbb{G}$ can be transformed into an algorithm that $(\tilde{\tau}, \epsilon)$-solves the ℓ-BDHI assumption in the same group $\mathbb{G}$, with the same advantage ϵ, and essentially the same running time $\tilde{\tau} \approx \tau$.

Suppose that $\mathbb{G}$ is a bilinear group of order p and generated by a random $w \in \mathbb{G}$. Let $\beta \in \mathbb{Z}_p^*$. For $\ell \geq 1$, the ℓ-th Bilinear Diffie-Hellman Inversion problem [5, 27] is usually stated as

$$\ell\text{-BDHI}: \qquad \text{given} \quad w, w^{\beta}, w^{(\beta^2)}, \ldots, w^{(\beta^{\ell})} \quad \text{compute} \quad e(w,w)^{\frac{1}{\beta}} .$$

Now, consider g and h, two random generators of $\mathbb{G}$, and let α be a random integer in $\mathbb{Z}_p^*$. For $\ell \geq 1$, the (modified) Weak BDHI problem from [6] is as follows:

$$\ell\text{-wBDHI*:} \qquad \text{given} \quad g, h, g^{\alpha}, g^{(\alpha^2)}, \ldots, g^{(\alpha^{\ell})} \quad \text{compute} \quad e(g,h)^{(\alpha^{\ell+1})} .$$

Any algorithm for ℓ-wBDHI* in $\mathbb{G}$ gives an algorithm for ℓ-BDHI in $\mathbb{G}$ with a tight reduction. Given an ℓ-BDHI problem instance $(w, w_1, \ldots, w_\ell)$, we build a ℓ-wBDHI* instance $(w_\ell, h, w_{\ell-1}, \ldots, w_1, w)$ where $h = w_\ell^r$ for some random exponent $r \in \mathbb{Z}_p^*$. If Z' be the solution to the ℓ-wBDHI* instance, then $Z = (Z')^{1/r}$ is the solution to the original ℓ-BDHI instance.

As usual, we say that the ℓ-wBDHI* assumption holds in $\mathbb{G}$ if no efficient algorithm can solve a random instance with non-negligible probability. For $i = 1, \ldots, \ell$, let $g_i = g^{(\alpha^i)} \in \mathbb{G}^*$. An algorithm $\mathcal{A}$ has advantage ϵ in solving the ℓ-wBDHI* problem in $\mathbb{G}$ if

$$\Pr\left[\mathcal{A}\big(g, h, g_1, \ldots, g_\ell\big) = e(g, h)^{(\alpha^{\ell+1})}\right] \geq \epsilon,$$

where the probability is over the random choice of generators g, h of $\mathbb{G}$, the random choice of α in $\mathbb{Z}_p^*$, and the random bits used by $\mathcal{A}$.

Definition A.1. The computational (t, ϵ, ℓ)-wBDHI* assumption holds in $\mathbb{G}$ if no t-time algorithm has advantage at least ϵ in solving a random instance of the ℓ-wBDHI* problem in $\mathbb{G}$.

A.1 Proving Forward Security

Proof of Theorem 5.1. Suppose that an algorithm $\mathcal{A}$ succeeds with advantage ϵ in time τ to produce a forgery in a random instance of the forward-secure signature scheme with $T = 2^\ell - 1$ time periods. Using $\mathcal{A}$, we build an algorithm $\mathcal{B}$ that solves the computational ℓ-wBDHI* problem in $\mathbb{G}$ with probability $\epsilon' \geq \epsilon/(4mqT)$ in time $\tau' \approx \tau$.

Thus, algorithm $\mathcal{B}$ is given as input a single random instance of the ℓ-wBDHI* problem, in the form of a tuple $(g, h, g_1, \ldots, g_\ell)$, where $g \in \mathbb{G}$ is a generator, and $g_i = g^{(\alpha^i)} \in \mathbb{G}$ for some undisclosed random $\alpha \in \mathbb{Z}_p^*$. The goal of $\mathcal{B}$ is to output a value $Z \in \mathbb{G}_t$ such that $Z = e(g, h)^{(\alpha^{\ell+1})}$. To do so, $\mathcal{B}$ will simulate the challenger against adversary $\mathcal{A}$ in the forward-secure existential forgery game. The process is as follows:

Initial Key Generation. To start the game, $\mathcal{B}$ must give $\mathcal{A}$ a verification key VK. Before that, $\mathcal{B}$ guesses the time period that $\mathcal{A}$ will choose to create a forgery. Let thus $\mathsf{ID}^* = (I_1^*, \ldots, I_\ell^*)$ be its identifier, randomly chosen by $\mathcal{B}$ in $\{0,1\}^\ell \setminus 0^\ell$. To create VK, algorithm $\mathcal{B}$ proceeds in two steps.

First, algorithm $\mathcal{B}$ picks a random integer $\beta \in \mathbb{Z}_p$, and constructs the triple $(g, g', g'') = (g, g^\alpha, g^{\beta+\alpha^\ell}) = (g, g_1, g^\beta g_\ell)$. This triple defines a CDH instance in $\mathbb{G}$, whose solution, $g^{\alpha\beta+\alpha^{\ell+1}}$, would let $\mathcal{B}$ calculate the solution to the wBDHI* problem as $Z = e(g,h)^{(\alpha^{\ell+1})} = e(g^{\alpha\beta+\alpha^{\ell+1}} g_1^{-\beta}, h)$. Algorithm $\mathcal{B}$ further selects a random integer $\rho \in \{0, \ldots, m\}$, followed by $m+1$ random integers $x_0, x_1, \ldots, x_m$ in the interval $\{0, \ldots, 2q-1\}$, and another $m+1$ integers $z_0, z_1, \ldots, z_m$ picked in $\mathbb{Z}_p$. It assigns $f_0 = (g'')^{x_0 - 2q\rho} g^{z_0}$, and $f_i = (g'')^{x_j} g^{z_j}$ for $i = 1, \ldots, m$.

Second, algorithm $\mathcal{B}$ picks $\ell+1$ random integers $\gamma_0, \gamma_1, \ldots, \gamma_\ell$ in $\mathbb{Z}_p$. Using the previously defined $g' = g_1 = g^\alpha$ and $g'' = g^\beta g_\ell = g^{\beta+\alpha^\ell}$, it computes $V = e(g', g'') = e(g, g)^{\alpha\beta+\alpha^{\ell+1}}$. This implicitly sets $\nu = \alpha\beta + \alpha^{\ell+1}$. The algorithm then assigns $h_0 = g^{\gamma_0} \cdot \prod_{i=1}^{\ell} g_{\ell-i+1}^{I_i^*}$, and $h_i = g^{\gamma_i} g_{\ell-i+1}^{-1}$ for $i = 1, \ldots, \ell$.

To finish the key generation, $\mathcal{B}$ picks an extra random integer, $\omega \in \mathbb{Z}_p$, lets $W = e(g, g)^\omega$, and gives $\mathcal{A}$ the verification key $\mathsf{VK} = (g, V, W, h_0, \ldots, h_\ell, f_0, \ldots, f_\ell)$. It also gives $\mathcal{A}$ the second factor decryption key $\mathsf{DecK} = g^\omega$ (even though it is not formally required at this stage). Finally, algorithm $\mathcal{B}$ sets up a public clock, $\mathsf{ID} \in \{0,1\}^\ell$, to keep track of the current time period; the clock is initially set to $\mathsf{ID} \leftarrow 0^{\ell-1}1$.

Interactive Query Phase. There are three types of queries. $\mathcal{A}$ can make as many Update queries as it wishes, and up to q Sign queries; these queries can be made in any order. $\mathcal{B}$ is to respond to each query before seeing the next one. $\mathcal{A}$ may then make a single final Corrupt query before moving to the Forgery phase. The queries and responses are handled as follows.

"Update" queries. $\mathcal{A}$ may request that $\mathcal{B}$ advance its clock to a new value $\mathsf{ID}' > \mathsf{ID}$, under the constraint that $\mathsf{ID}' \in \{0,1\}^\ell$. To respond, $\mathcal{B}$ updates $\mathsf{ID} \leftarrow \mathsf{ID}'$ and acknowledges.

"Sign" queries. $\mathcal{A}$ may request that $\mathcal{B}$ sign a message $M \in \{0,1\}^m$ of $\mathcal{A}$'s choosing, using the key for the current time period as indicated by $\mathcal{B}$'s clock, ID. Algorithm $\mathcal{B}$ responds as follows.

If $\mathsf{ID} \neq \mathsf{ID}^*$, algorithm $\mathcal{B}$ first generates a (partial) signing key that will let it create a signature using the regular *Sign* procedure. Recall from the scheme description that *Sign* only necessitates $K_{\ell+1} = (a_0, a_1)$ from $\mathsf{EncSK}_{\mathsf{ID}}$. Since $k_{\ell+1} = \mathsf{sibling}(\ell+1, \mathsf{ID}) = \mathsf{ID} \neq \mathsf{ID}^* = \mathsf{sibling}(\ell+1, \mathsf{ID}^*)$, algorithm $\mathcal{B}$ is able to compute $K_{\ell+1}$ even though it may not be able to assemble the full key $\mathsf{EncSK}_{\mathsf{ID}}$. The process to compute $K_{\ell+1}$ is essentially as described below, when answering the "Corrupt" query.

If, $\mathsf{ID} = \mathsf{ID}^*$, however, algorithm $\mathcal{B}$ must proceed differently. Let $M = (\mu_1, \ldots, \mu_m) \in \{0,1\}^m$ be the message to be signed. Let $\mathcal{B}$ compute $D = -2q\rho + x_0 + \sum_{i=1}^m x_i \mu_i$ and $J = z_0 + \sum_{i=1}^m z_i \mu_i$. If $D \equiv 0 \pmod p$, then $\mathcal{B}$ cannot produce the requested signature and must abort. Otherwise, $\mathcal{B}$ picks two random integers $r, s \in \mathbb{Z}_p$, lets $h = h_0 \cdot \prod_{j=1}^\ell h_j^{\mathsf{bit}(\mathsf{ID}^*, j)}$, and outputs the signature

$$S = (\mathsf{ID}, s_0, s_1, s_2) = \Big(\; \mathsf{ID}, \;\; g_1^{-J/D} \cdot h^r \cdot (f_0 \prod_{i=1}^m f_i^{\mu_i})^s, \;\; g^r, \;\; g_1^{1/D} g^s \; \Big).$$

Let us substitute the expression $\alpha/D + \tilde{s}$ for the random exponent s. We successively rewrite $s_0 = g_1^{-J/D} \cdot h^r \cdot ((g'')^D g^J)^s = (g^\alpha)^{-J/D} \cdot h^r \cdot (g'')^{\alpha + D\tilde{s}} g^{\alpha J/D + J\tilde{s}} = g^{\alpha(\beta+\alpha^\ell)} h^r (g'')^{D\tilde{s}} g^{J\tilde{s}} = g^\nu h^r ((g'')^D g^J)^{\tilde{s}} = g^\nu (h_0 \prod_{j=1}^\ell h_j^{\mathsf{bit}(\mathsf{ID},j)})^r (f_0 \prod_{i=1}^m f_i^{\mathsf{bit}(M,i)})^{\tilde{s}}$, and similarly, $s_2 = g_1^{1/D} g^{a/D+\tilde{s}} = g^{\tilde{s}}$. This shows that S is a valid signature with the correct distribution, as the exponents r and $\tilde{s}$ are uniform over $\mathbb{Z}_p$.

"Corrupt" query. $\mathcal{A}$ may make a final request that $\mathcal{B}$ reveal the encrypted signing key $\mathsf{EncSK}_{\mathsf{ID}}$ for the current value of the clock ID. (The decryption key DecK was already given to $\mathcal{A}$ during key generation.)

If $\mathsf{ID} \leq \mathsf{ID}^*$, algorithm $\mathcal{B}$ terminates its execution since it is unable to answer the query. Otherwise, algorithm $\mathcal{B}$ constructs a key $\mathsf{EncSK}_{\mathsf{ID}} = (\mathsf{ID}, K_1, \ldots, K_{\ell+1})$, where each component is computed as follows. For $j = 1, \ldots, \ell+1$, we define

$$\mathsf{sibling}(j, \mathsf{ID}) = \begin{cases} (\mathrm{I}_1, \ldots, \mathrm{I}_{j-1}, 1) & \text{if } j < \ell+1 \text{ and } \mathrm{I}_j = 0 \;, \\ \perp & \text{if } j < \ell+1 \text{ and } \mathrm{I}_k = 1 \;, \\ \mathsf{ID} & \text{if } j = \ell+1 \;, \end{cases}$$

as in the scheme description, and let $k_j = \mathsf{sibling}(j, \mathsf{ID})$. If $k_j = \perp$, then we let $\mathcal{B}$ assign $K_j = \perp$. Otherwise, we know that k_j is a bit string of length $j' = \min\{j, \ell\}$, and that it is not a prefix of ID. For a binary string S, let us denote by $S_{|j'}$ the first j' bits of S. It is easy to see that k_j is not a prefix of ID^* either because, by definition of $\mathsf{sibling}$, we have $k_j > \mathsf{ID}_{|j'}$, and in the current context we have $\mathsf{ID} > \mathsf{ID}^*$, and thus $\mathsf{ID}_{|j'} \geq \mathsf{ID}^*_{|j'}$. Hence, $k_j \neq \mathsf{ID}^*_{|j'}$, and therefore there exists an index $\theta \leq j$ such that $\mathsf{bit}(k_j, \theta) \neq \mathsf{bit}(\mathsf{ID}^*, \theta)$. W.l.o.g., suppose that θ is the smallest such index. Recall that $\mathsf{ID}^* = (\mathrm{I}^*_1, \ldots, \mathrm{I}^*_\ell)$. We similarly expand k_j as $(\mathrm{I}'_1, \ldots, \mathrm{I}'_j)$. Note that $\mathrm{I}'_\theta \neq \mathrm{I}^*_\theta$.

$\mathcal{B}$ generates K_j in two steps. First, it creates a "prototype" K'_j based on the first θ bits of k_j. Let $\mathcal{B}$ pick a random integer $\tilde{r}$ in $\mathbb{Z}_p$, and pose $r = \frac{\alpha^\theta}{(\mathrm{I}'_\theta - \mathrm{I}^*_\theta)} + \tilde{r} \in \mathbb{Z}_p$. The prototype is given as

$$K'_j = (a_0, a_1, b_{\theta+1}, \ldots, b_\ell) = \left(g^{\nu+\omega} \cdot (h_0 \cdot h_1^{\mathrm{I}'_1} \cdot \ldots \cdot h_\theta^{\mathrm{I}'_\theta})^r,\ \ g^r,\ \ h_{\theta+1}^r,\ \ \ldots,\ \ h_\ell^r \right) \ .$$

We show that $\mathcal{B}$ can compute K'_j without knowing r or ν. Recall that $g_i^{(\alpha^j)} = g_{i+j}$ for any i and j. For a_0, we successively get

$$\begin{aligned}
a_0 &= g^{\nu+\omega} \cdot (h_0 \cdot h_1^{\mathrm{I}'_1} \cdots h_\theta^{\mathrm{I}'_\theta})^r \\
&= g^{\nu+\omega} \cdot \left(g^{\gamma_0 + \sum_{i=1}^{\theta} \mathrm{I}'_i \gamma_i} \cdot \left(\prod_{i=1}^{\theta-1} g_{\ell-i+1}^{(\mathrm{I}^*_i - \mathrm{I}'_i)} \right) \cdot g_{\ell-\theta+1}^{(\mathrm{I}^*_\theta - \mathrm{I}'_\theta)} \cdot \left(\prod_{i=\theta+1}^{\ell} g_{\ell-i+1}^{\mathrm{I}^*_i} \right) \right)^r \\
&= g^{\alpha\beta + \alpha^{\ell+1}} \cdot g_{\ell-\theta+1}^{-\alpha^\theta} \cdot g_{\ell-\theta+1}^{(\mathrm{I}^*_\theta - \mathrm{I}'_\theta)\tilde{r}} \cdot g^\omega \cdot \left(g^{\gamma_0 + \sum_{i=1}^{\theta} \mathrm{I}'_i \gamma_i} \cdot \underbrace{\left(\prod_{i=1}^{\theta-1} g_{\ell-i+1}^{(\mathrm{I}^*_i - \mathrm{I}'_i)} \right)}_{=1} \cdot \left(\prod_{i=\theta+1}^{\ell} g_{\ell-i+1}^{\mathrm{I}^*_i} \right) \right)^r \\
&= g^{\alpha\beta} \cdot g_{\ell-\theta+1}^{(\mathrm{I}^*_\theta - \mathrm{I}'_\theta)\tilde{r}} \cdot g^\omega \cdot \left(g^{\gamma_0 + \sum_{i=1}^{\theta} \mathrm{I}'_i \gamma_i} \cdot \prod_{i=\theta+1}^{\ell} g_{\ell-i+1}^{\mathrm{I}^*_i} \right)^r \ .
\end{aligned}$$

The $\alpha^{\ell+1}$-th powers of g are unknown to $\mathcal{B}$, but they cancel each other out. In addition, the second factor in the big parentheses vanishes because $\mathrm{I}'_i = \mathrm{I}^*_i$ for $1 \le i < \theta$. It is easy to verify that all of what remains can be expressed in terms of the values of g_i at $\mathcal{B}$'s disposal. $\mathcal{B}$ can thus calculate a_0. The remaining components of the prototype are easy to compute. Considering a_1, we have

$$a_1 = g^r = (g_\theta)^{1/(\mathrm{I}'_\theta - \mathrm{I}^*_\theta)} \cdot g^{\tilde{r}} \ ,$$

which $\mathcal{B}$ can compute. For $b_{\theta+1}$, we have

$$b_{\theta+1} = h_{\theta+1}^r = (g_\theta)^{-\gamma_{\theta+1}/(\mathrm{I}'_\theta - \mathrm{I}^*_\theta)} \cdot g^{\gamma_{\theta+1}\tilde{r}} \cdot (g_\ell)^{-1/(\mathrm{I}'_\theta - \mathrm{I}^*_\theta)} \cdot (g_{\ell-\theta})^{-\tilde{r}} \ .$$

The remaining components are similarly obtained, since none of them involves the term $g_{\ell+1}$.

The second step for $\mathcal{B}$, once K'_j has been calculated, is to transform $K'_j = (a_0, a_1, b_{\theta+1}, \ldots, b_\ell)$ into K_j as in the *Update* procedure. To do so, $\mathcal{B}$ picks a random $r' \in \mathbb{Z}_p$, and computes

$$K_j = \left(a_0 \cdot \left(b_{\theta+1}^{\mathrm{I}'_{\theta+1}} \cdot \ldots \cdot b_j^{\mathrm{I}'_j} \right) \cdot \left(h_0 \cdot h_1^{\mathrm{I}'_1} \cdot \ldots \cdot h_j^{\mathrm{I}'_j} \right)^{r'},\ \ a_1 \cdot g^{r'},\ \ b_{j+1} \cdot h_{j+1}^{r'},\ \ \ldots,\ \ b_\ell \cdot h_\ell^{r'} \right) \ .$$

Once this two-step process has been performed for each $j = 1, \ldots, \ell+1$, $\mathcal{B}$ ends up with $\mathsf{EncSK}_{\mathsf{ID}}$, which it gives to $\mathcal{A}$, as required.

Forgery and Reduction. When it is done with the query phase, $\mathcal{A}$ outputs an existential forgery "in the past". The forgery consists of a message $M^* = (\mu^*_1 \ldots \mu^*_m) \in \{0,1\}^m$ and a signature $S^* = (\mathsf{ID}^{**}, s^*_0, s^*_1, s^*_2)$, under the constraint that $\mathcal{A}$ must not have previously requested a signature on M^* when the clock read ID^{**}. From this point on, there is no further interaction with $\mathcal{A}$.

If $\mathsf{ID}^{**} \neq \mathsf{ID}^*$, then $\mathcal{B}$ has failed as it will not be able to complete the reduction. However, if $\mathcal{B}$ had correctly guessed $\mathsf{ID}^{**} = \mathsf{ID}^*$, it may proceed as follows. Let $D^* = -2\rho q + x_0 + \sum_{i=1}^{m} x_i \mu^*_i$. If

$D^* \not\equiv 0 \pmod p$, algorithm $\mathcal{B}$ aborts. Otherwise, let $J^* = z_0 + \sum_{i=1}^m z_i \mu_i^*$ and $H^* = \gamma_0 + \sum_{j=1}^{\ell} \gamma_j \mathrm{I}_j^*$. For some integers r and s in $\mathbb{Z}_p$, the forgery $S^* = (\mathsf{ID}^*, s_0^*, s_1^*, s_2^*)$ is necessarily of the form

$$S^* = \Big(\mathsf{ID}^*,\ g^{\nu}(h_0 \prod_{j=1}^{\ell} h_j^{\mathsf{bit}(\mathsf{ID}^*,j)})^r (f_0 \prod_{i=1}^{m} f_i^{\mathsf{bit}(M^*,i)})^s,\ g^r,\ g^s\Big) = \big(\mathsf{ID}^*, g^{\alpha\beta+\alpha^{\ell+1}+H^*r+J^*s}, g^r, g^s\big)\ .$$

To obtain $g^{\alpha\beta+\alpha^{\ell+1}}$ and thereby solve the synthetic CDH instance (g, g', g''), algorithm $\mathcal{B}$ computes $(s_2^*)^{-J^*}(s_1^*)^{-H^*}(s_0^*) = g^{\alpha\beta+\alpha^{\ell+1}} = g^{\alpha\beta+\alpha^{\ell+1}}$. As mentioned earlier, $\mathcal{B}$ is now able to solve the original wBDHI* instance: $Z = e(g,h)^{(\alpha^{\ell+1})} = e(g^{\alpha\beta+\alpha^{\ell+1}} g_1^{-\beta}, h)$.

Success Probability. To conclude this proof, we give a lower bound on the efficiency of the reduction. To succeed, the guess ID^* must be correct, which has a prior probability of $1/T = 1/(2^\ell - 1)$. In addition, $\mathcal{B}$ must not abort upon answering signature queries. It is always able to sign any message at a time $\mathsf{ID} \neq \mathsf{ID}^*$. At time period ID^*, however, there is a small failure probability $\leq 1/(2q)$ for each new signature request. Since the number of signature queries is bounded by q, by the union bound it follows that $\mathcal{B}$ will be able to satisfy all the requests with conditional probability at least 1/2. Last, for the forgery phase to succeed, it is necessary that $D^* \equiv 0 \pmod p$, which happens with conditional probability at least $1/(2mq)$. Therefore, if $\mathcal{A}$ makes at most q signature queries and breaks the forward-secure signature scheme with probability ϵ, then $\mathcal{B}$ solves the wBDHI* problem with probability $\epsilon' \geq \epsilon/(4mqT)$.

This concludes the proof of Theorem 5.1. ☐

A.2 Proving Update Security

Proof of Theorem 5.2. Consider an instantiation of our signature scheme with $T = 2^\ell - 1$ time periods, and suppose that there is an algorithm $\mathcal{A}$ that runs in time τ and wins the update security game with advantage ϵ while making no more than q signature queries. Then we show how to construct an algorithm $\mathcal{B}$ that solves the computational Diffie-Hellman problem in the bilinear group $\mathbb{G}$ with probability $\epsilon' \geq \epsilon/(4qm)$ in time $\tau' \approx \tau$.

Algorithm $\mathcal{B}$ is given an instance (g, g', g'') of the CDH problem in $\mathbb{G}$, where $g \in \mathbb{G}$ is a generator, and $g' = g^\alpha$ and $g'' = g^\beta$ for undisclosed random $\alpha, \beta \in \mathbb{Z}_p^*$. Its goal is to compute $Z = g^{\alpha\beta} \in \mathbb{G}$. Algorithm $\mathcal{B}$ works by simulating an attack environment to $\mathcal{A}$ per the update security game. The process is as follows:

Initial Key Generation. To launch the game, $\mathcal{B}$ must give $\mathcal{A}$ a verification key VK and an initial encrypted signing key $\mathsf{EncSK}_{\mathsf{ID}}$ for $\mathsf{ID} = 0^{\ell-1}1$.

Algorithm $\mathcal{B}$ starts by selecting a random integer $\delta \in \mathbb{Z}_p$. It computes $V = e(g', g'') = e(g,g)^{\alpha\beta}$, and assigns $W = e(g,g)^\delta \cdot V^{-1}$. This implicitly defines $\nu = \alpha\beta \pmod p$ and $\omega = \delta - \nu \pmod p$.

Algorithm $\mathcal{B}$ then selects a random integer $\rho \in \{0, \ldots, m\}$, and picks $m+1$ random integers $x_0, x_1, \ldots, x_m$ in the interval $\{0, \ldots, 2q-1\}$. It also picks $m+1$ integers $z_0, z_1, \ldots, z_m$ in $\mathbb{Z}_p$, and assigns $f_0 = (g'')^{x_0 - 2q\rho} g^{z_0}$, and $f_i = (g'')^{x_j} g^{z_j}$ for $i = 1, \ldots, m$. Finally, it picks $\ell+1$ random integers $\gamma_0, \gamma_1, \ldots, \gamma_\ell$ in $\mathbb{Z}_p$, and assigns $h_0 = g^{\gamma_0}$, and $h_i = g^{\gamma_i}$ for $i = 1, \ldots, \ell$.

The clock is set to $\mathsf{ID} \leftarrow 0^{\ell-1}1$, and the verification key is $\mathsf{VK} = (g, V, W, h_0, \ldots, h_\ell, f_0, \ldots, f_\ell)$. The initial encrypted signing key $\mathsf{EncSK}_{\mathsf{ID}}$ is computed as in the scheme itself: even though $\mathcal{B}$ does not know either ν or ω, it can compute $g^{\nu+\omega} = g^\delta$, and the rest is known. $\mathcal{B}$ gives to $\mathcal{A}$ the keys VK and $\mathsf{EncSK}_{\mathsf{ID}}$. Nobody gets DecK.

Interactive Query Phase. There are two types of queries: Update and Sign. $\mathcal{A}$ can make as many Update queries as it wants, and at most q Sign queries, in any order. $\mathcal{B}$ must respond at once to each query.

"Update" queries. $\mathcal{A}$ can request $\mathcal{B}$ to advance the clock to any new value ID' of its choice, provided that $\mathsf{ID}' > \mathsf{ID}$ and that $\mathsf{ID}' \in \{0,1\}^\ell$. $\mathcal{B}$ responds by updating $\mathsf{ID} \leftarrow \mathsf{ID}'$.

"Sign" queries. $\mathcal{A}$ may request that $\mathcal{B}$ sign a message $M \in \{0,1\}^m$ of $\mathcal{A}$'s choosing, using an encrypted signing key EncSK that is also supplied by $\mathcal{A}$. The time period identifier embedded in the key must match that indicated by $\mathcal{B}$'s clock, ID. Algorithm $\mathcal{B}$ responds as follows.

The first task for $\mathcal{B}$ is to determine whether the signing key is valid, at least for time period ID, by testing whether $1 = V \cdot W \cdot e(a_0, g^{-1}) \cdot e\big(a_1, h_0 \prod_{i=1}^{\ell} h_i^{\mathsf{bit}(i,\mathsf{ID})}\big)$ in $\mathbb{G}_t$, as specified in the actual scheme. If the test fails, the key is demonstrably invalid (since anyone can perform the same test), and thus $\mathcal{B}$ has a justification to respond with $\perp$.

If the test passes, then EncSK is valid for the current time period, and $\mathcal{B}$ should produce a signature. Instead of using EncSK, which it cannot decrypt, $\mathcal{B}$ will create a signature from scratch, that will be have the same uniform distribution over valid signatures as if the *Sign* procedure had been used. This is fine because the specification of *Sign* clearly shows that the signature distribution is conditionally independent of EncSK (if valid), for a given VK, time, and message.

Let thus $M = (\mu_1, \ldots, \mu_m) \in \{0,1\}^m$ be the message to be signed at time ID. Let $\mathcal{B}$ compute $D = -2q\rho + x_0 + \sum_{i=1}^m x_i \mu_i$ and $J = z_0 + \sum_{i=1}^m z_i \mu_i$. If $D \equiv 0 \pmod p$, then $\mathcal{B}$ must abort, being unable to respond. Otherwise, $\mathcal{B}$ picks two random integers $r, s \in \mathbb{Z}_p$, lets $h = h_0 \cdot \prod_{j=1}^{\ell} h_j^{\mathsf{bit}(\mathsf{ID}^*,j)}$, and outputs the signature

$$S = (\mathsf{ID}, s_0, s_1, s_2) = \Big(\ \mathsf{ID}, \ \ (g')^{-J/D} \cdot h^r \cdot (f_0 \prod_{i=1}^m f_i^{\mu_i})^s, \ \ g^r, \ \ (g')^{1/D} g^s \ \Big).$$

To see that this is a good uniformly distributed signature, we substitute $s = \alpha/D + \tilde{s}$, then rewrite $s_0 = (g')^{-J/D} \cdot h^r \cdot ((g'')^D g^J)^s = (g^\alpha)^{-J/D} \cdot h^r \cdot (g'')^{\alpha + D\tilde{s}} g^{\alpha J/D + J\tilde{s}} = g^{\alpha\beta} h^r (g'')^{D\tilde{s}} g^{J\tilde{s}} = g^\nu h^r ((g'')^D g^J)^{\tilde{s}} = g^\nu (h_0 \prod_{j=1}^{\ell} h_j^{\mathsf{bit}(\mathsf{ID},j)})^r (f_0 \prod_{i=1}^m f_i^{\mathsf{bit}(M,i)})^{\tilde{s}}$, and similarly, $s_2 = (g')^{1/D} g^{a/D + \tilde{s}} = g^{\tilde{s}}$. This shows that S is a valid signature whose randomization exponents r and $\tilde{s}$ are independent and uniform in $\mathbb{Z}_p$.

Forgery and Reduction. When it is done with the query phase, $\mathcal{A}$ outputs an existential forgery. The forgery consists of a message $M^* = (\mu_1^* \ldots \mu_m^*) \in \{0,1\}^m$ and a signature $S^* = (\mathsf{ID}^*, s_0^*, s_1^*, s_2^*)$, under the constraint that $\mathcal{A}$ must not have previously requested a signature on M^* at any time. There are no further interactions with $\mathcal{A}$ from this point on.

Compute $D^* = -2\rho q + x_0 + \sum_{i=1}^m x_i \mu_i^*$. If $D^* \not\equiv 0 \pmod p$, algorithm $\mathcal{B}$ must abort. Otherwise, let $J^* = z_0 + \sum_{i=1}^m z_i \mu_i^*$ and $H^* = \gamma_0 + \sum_{j=1}^{\ell} \gamma_j \mathrm{I}_j^*$. For some integers r and s in $\mathbb{Z}_p$, the forgery $S^* = (\mathsf{ID}^*, s_0^*, s_1^*, s_2^*)$ can be expressed as

$$S^* = \Big(\mathsf{ID}^*, \ g^\nu (h_0 \prod_{j=1}^{\ell} h_j^{\mathsf{bit}(\mathsf{ID}^*,j)})^r (f_0 \prod_{i=1}^m f_i^{\mathsf{bit}(M^*,i)})^s, \ g^r, \ g^s\Big) = \big(\mathsf{ID}^*, g^{\alpha\beta + H^* r + J^* s}, g^r, g^s\big) \ .$$

To extract $g^{\alpha\beta}$ and thereby solve the given CDH instance (g, g', g'') in $\mathbb{G}$, algorithm $\mathcal{B}$ computes $(s_2^*)^{-J^*} (s_1^*)^{-H^*} (s_0^*) = g^{\alpha\beta} = g^{\alpha\beta}$, and outputs the result.

Success Probability. It remains to bound the efficiency of the reduction. First, $\mathcal{B}$ must not abort at any time when answering signature queries. For each new message, the probability of aborting is at most $1/(2q)$; it is 0 for any repeat message even if the time period has changed. Since $\mathcal{A}$ makes requests on at most q unique messages, $\mathcal{B}$ will complete the query phase with probability at least $1/2$. Second, for the forgery phase to succeed, we must have $D^* \equiv 0 \pmod{p}$, which for a new message M^* happens with conditional probability at least $1/(2mq)$. In summary, if $\mathcal{A}$ wins the update security game with probability ϵ, then $\mathcal{B}$ solves the CDH problem in $\mathbb{G}$ with probability $\epsilon' \geq \epsilon/(4mq)$. $\square$

Observe that the success probability argument in this proof depends on the adversary not querying on the forgery message M^* at any other time period. This can easily be realized by embedding the time period into the message itself. It is easy to accommodate in the construction by letting $M = H(\mathsf{ID}\|\mathrm{Msg})$ or $M = R\|H_R(\mathsf{ID}\|\mathrm{Msg})$, where Msg is the true message, and H is either a collision-resistant hash function or a UOWHF family indexed by R.