Pixel: Multi-signatures for Consensus

Manu Drijvers
DFINITY

Sergey Gorbunov
Algorand and University of Waterloo

Gregory Neven
DFINITY

Hoeteck Wee*
Algorand and CNRS, ENS, PSL

Abstract

In Proof-of-Stake (PoS) and permissioned blockchains, a committee of verifiers agrees and sign every new block of transactions. These blocks are validated, propagated, and stored by all users in the network. However, posterior corruptions pose a common threat to these designs, because the adversary can corrupt committee verifiers after they certified a block and use their signing keys to certify a different block. Designing efficient and secure digital signatures for use in PoS blockchains can substantially reduce bandwidth, storage and computing requirements from nodes, thereby enabling more efficient applications.

We present Pixel, a pairing-based forward-secure multi-signature scheme optimized for use in blockchains, that achieves substantial savings in bandwidth, storage requirements, and verification effort. Pixel signatures consist of two group elements, regardless of the number of signers, can be verified using three pairings and one exponentiation, and support non-interactive aggregation of individual signatures into a multi-signature. Pixel signatures are also forward-secure and let signers evolve their keys over time, such that new keys cannot be used to sign on old blocks, protecting against posterior corruptions attacks on blockchains. We show how to integrate Pixel into any PoS blockchain. Next, we evaluate Pixel in a real-world PoS blockchain implementation, showing that it yields notable savings in storage, bandwidth, and block verification time. In particular, Pixel signatures reduce the size of blocks with 1500 transactions by 35% and reduce block verification time by 38%.

1 Introduction

Blockchain technologies are quickly gaining popularity for payments, financial applications, and other distributed applications. A blockchain is an append-only public ledger that is maintained and verified by distributed nodes. At the core of the blockchain is a consensus mechanism that allows nodes to agree on changes to the ledger, while ensuring that changes once confirmed cannot be altered.

In the first generation of blockchain implementations, such as Bitcoin, Ethereum, Litecoin, the nodes with the largest computational resources choose the next block. These implementations suffer from many known inefficiencies, low throughput, and high transaction latency [18, 28, 52]. To overcome these problems, the current generation of blockchain implementations such as Algorand, Cardano, Ethereum Casper, and Dfinity turn to proofs of stake (PoS), where nodes with larger stakes in the system—as measured, for instance, by the amount of money in their account—are more likely to participate in choosing the next block [22, 25, 31, 34, 36, 41, 50]. Permissioned blockchains such as Ripple [57] and Hyperledger Fabric [4] take yet another approach, sacrificing openness for efficiency by limiting participation in the network to a selected set of nodes.

All PoS-based blockchains, as well as permissioned ones, have a common structure where the nodes run a consensus sub-protocol to agree on the next block to be added to the ledger. Such a consensus protocol usually requires nodes to inspect block proposals and express their agreement by digitally signing acceptable proposals. When a node sees sufficiently many signatures from other nodes on a particular block, it appends the block to its view of the ledger.

Because the consensus protocol often involves thousands of nodes working together to reach consensus, efficiency of the signature scheme is of paramount importance. Moreover, to enable outsiders to efficiently verify the validity of the chain, signatures should be compact to transmit and fast to verify. Multi-signatures [37] have been found particularly useful for this task, as they enable many signers to create a compact and efficiently verifiable signature on a common message [16, 42, 61, 62].

The Problem of Posterior Corruptions. Chain integrity in a PoS blockchain relies on the assumption that the adversary controls less than a certain threshold (e.g., a third) of the total stake; an adversary controlling more than that fraction
may be able to fork the chain, i.e., present two different but equally valid versions of the ledger. Because the distribution of stake changes over time, however, the real assumption behind chain integrity is not just that the adversary currently controls less than a threshold of the stake, but that he never did so at any time in the past.

This assumption becomes particularly problematic if stake control is demonstrated through possession of signature keys, as is the case in many PoS and permissioned blockchains. Indeed, one could expect current stakeholders to properly protect their stake-holding keys, but they may not continue to do so forever, especially after selling their stake. Nevertheless, without additional precautions, an adversary who obtains keys that represent a substantial fraction of stake at some point in the past can compromise the ledger at any point in the future. The problem is further aggravated in efficient blockchains that delegate signing rights to a small committee of stakeholders, because the adversary can gain control of the chain after corrupting a majority of the committee members.

Referred to by different authors as long-range attacks [21], costless simulation [55], and posterior corruptions [12], this problem is best addressed through the use of forward-secure signatures [3,9,43,48]. Here, each signature is associated with the current time period, and a user’s secret key can be updated in such a way that it can only be used to sign messages for future time periods, not previous ones. An adversary that corrupts an honest node can therefore not use the compromised key material to create forks in the past of the chain.

1.1 Our Results

We present the Pixel signature scheme, which is a pairing-based forward-secure multi-signature scheme for use in PoS-based blockchains that achieves substantial savings in bandwidth and storage requirements. To support a total of $T$ time periods and a committee of size $N$, the multi-signature comprises just two group elements and verification requires only three pairings, one exponentiation, and $N-1$ multiplications. Pixel signatures are almost as efficient as BLS multi-signatures, as depicted in Figure 1, but also satisfy forward-security; moreover, like in BLS multi-signatures, anybody can non-interactively aggregate individual signatures into a multi-signature.

Our construction builds on prior forward-secure signatures based on hierarchical identity-based encryption (HIBE) [15, 19,23,27] and adds the ability to securely aggregate signatures on the same message as well as to generate public parameters without trusted set-up.

We achieve security in the random oracle model under a variant of the bilinear Diffie-Hellman inversion assumption [11,15]. At a very high level, the use of HIBE techniques allows us to compress $O(\log T)$ group elements in a tree-based forward-secure signature into two group elements, and secure aggregation allows us to compress $N$ signatures under

$N$ public keys into a single multi-signature of the same size as a single signature.

To validate Pixel’s design, we compared the performance of a Rust implementation [2] of Pixel with previous forward-secure tree-based solutions. We show how to integrate Pixel into any PoS blockchain. Next, we evaluate Pixel on the Algorand blockchain, showing that it yields notable savings in storage, bandwidth, and block verification time. Our experimental results show that Pixel is efficient as a stand-alone primitive and in use in blockchains. For instance, compared to a set of $N = 1500$ tree-based forward-secure signatures (for $T = 2^{32}$) at 128-bit security level, a single Pixel signature that can authenticate the entire set is 2667x smaller and can be verified 40x faster (c.f. Tables 1 and 3). Pixel signatures reduce the size of Algorand blocks with 1500 transactions by $\approx 35\%$ and reduce block verification time by $\approx 38\%$ (c.f. Figures 4 and 5).

1.2 Related Work

Multi-signatures can be used to generate a single short signature validates that a message $m$ was signed by $N$ different parties [6, 10, 14, 33, 37, 45, 46, 51, 53]. Multi-signatures based on the BLS signature scheme [14, 16, 17, 56] are particularly well-suited to the distributed setting of PoS blockchains as no communication is required between the signers; anybody can aggregate individual signatures into a multi-signature. However, these signatures are not forward-secure.

Tree-based forward-secure signatures [9, 38, 43, 48] can be used to meet the security requirements, but they are not very efficient in an $N$-signer setting because all existing constructions have signature size at least $O(N\log T)$ group elements, where $T$ is an upper bound on the number of time periods. Some schemes derived from hierarchical identity-based encryption (HIBE) [15, 19,23] can bring that down to $O(N)$ group elements, which is still linear in the number of signers. The only forward-secure multi-signature schemes that appeared in the literature so far have public key length linear in the number of time periods $T$ [47] or require interaction between the signers to produce a multi-signature [58], neither of which is desirable in a blockchain scenario. The forward-secure multi-signature scheme of Yu et al. [64] has signature length linear in the number of signers, so is not really a multi-signature scheme.

Combining the generic tree-based forward-secure signature scheme of Bellare-Miner [9] with BLS multi-signatures [14, 17] gives some savings, but still requires $O(T)$ “certificates” to be included in each multi-signature. Batch verification [8] can be used to speed up verification of the certificates to some extent, but does not give us any space savings. Compared with existing tree-based forward-secure signatures in [9,38,43,48], our savings are two-fold:

- we reduce the size of the signature set for $N$ commit-
19, 23, 27] and adds the ability to securely aggregate signatures without trusted set-up.

The rest of this paper is organized as follows:

- In Section 2, we give a high level technical description of our new pairing-based forward-secure multi-signature scheme.
- In Sections 4 and 5, we describe the scheme in details. We prove the security of the construction in the random oracle model under a variant of a bilinear Diffie-Hellman inversion problem.
- In Section 6, we explain how to apply Pixel to PoS blockchains to solve posterior corruptions.
- In Section 7, we evaluate the efficiency savings for storage, bandwidth, and block verification time from using Pixel on the Algorand PoS blockchain.
- In Section 8, we describe various extensions to the scheme.
- In Appendix C, we perform theoretical efficiency analysis of Pixel.

## 1.3 Paper Organization

The rest of this paper is organized as follows:

- In Section 2, we give a high level technical description of our new pairing-based forward-secure multi-signature scheme.
- In Sections 4 and 5, we describe the scheme in details. We prove the security of the construction in the random oracle model under a variant of a bilinear Diffie-Hellman inversion problem.
- In Section 6, we explain how to apply Pixel to PoS blockchains to solve posterior corruptions.
- In Section 7, we evaluate the efficiency savings for storage, bandwidth, and block verification time from using Pixel on the Algorand PoS blockchain.
- In Section 8, we describe various extensions to the scheme.
- In Appendix C, we perform theoretical efficiency analysis of Pixel.

### 2 Technical Overview

Our construction builds on prior forward-secure signatures based on hierarchical identity-based encryption (HIBE) [15, 19, 23, 27] and adds the ability to securely aggregate signatures on the same message as well as to generate public parameters without trusted set-up.

#### Overview of our scheme.

Starting with a bilinear group $(G_1, G_2, G_t)$ with $e : G_1 \times G_2 \rightarrow G_t$ of prime order $q$ and generators $g_1, g_2$ for $G_1, G_2$ respectively, a signature on $M \in \mathbb{Z}_q$ at time $t$ under public key $g^x$ is of the form:

$$\sigma = (\sigma', \sigma'') = (h^{\cdot} F(t, M), g^x_2) \in G_1 \times G_2$$

where the function $F(t, M)$ can be computed with some public parameters (two group elements in $G_1$ in addition to $h \in G_1$) and $r$ is fresh randomness used for signing. Verification relies on the relation:

$$e(\sigma', g_2) = e(h, y) \cdot e(F(t, M), \sigma'')$$

and completeness follows directly:

$$e(\sigma', g_2') = e(h^{r'}, F(t, M), g_2')$$

Note that $e(h, y)$ can be precomputed to save verification computation.

Given $N$ signatures $\sigma_1, \ldots, \sigma_N \in G_1 \times G_2$ on the same message $M$ at time $t$ under $N$ public keys $g^x_2^{1}, \ldots, g^x_2^{N}$, we can produce a multi-signature $\Sigma$ on $M$ by computing the coordinate-wise product of $\sigma_1, \ldots, \sigma_N$. Concretely, if $\sigma_i = (h^{x_i} \cdot F(t, M)^{y_i}, g^x_2^{i})$, then

$$\Sigma = (h^{x_1 + \cdots + x_N} \cdot F(t, M)^{r'}, g^x_2^{r'})$$

where $r' = r_1 + \cdots + r_N$. To verify $\Sigma$, we first compute a single aggregate public key that is a compressed version of all $N$ individual public keys

$$apk \leftarrow y_1 \cdot \cdots \cdot y_N,$$

and verify $\Sigma$ against $apk$ using the standard verification equation.

#### How to generate and update keys.

To complete this overview, we describe a simplified version of the secret keys and update mechanism, where the secret keys are of size $O(T)$

| scheme | key update | sign | verify | $|\sigma|$ | $|pk|$ | $|sk|$ | forward security |
|--------|------------|------|--------|----------|------|------|-----------------|
| BLS multi-signatures [14, 16, 56] | – | 1 exp | 2 pair | 1 | 1 | $O(1)$ | no |
| Pixel multi-signatures (this work) | 2 exp | 4 exp | 3 pair + 1 exp | 2 | 1 | $O((\log T)^2)$ | yes |

1 Each tree-based signature comprise $O(\log T)$ group elements corresponding to a path in a tree of depth $\log T$ (see Section 7 for details), and there are $N$ such signatures, one for which committee member.
instead of $O((\log T)^2)$. The construction exploits the fact that the function $F$ satisfies

$$F(t, M) = F(t, 0) \cdot F^M$$

for some constant $F'$. This means that in order to sign messages at time $t$, it suffices to know

$$\tilde{sk}_t = \{ h^t \cdot F(t, 0)' , F'^t, g^t_2 \}$$

from which we can compute $(h^t \cdot F(t, M)', g^t_2)$.

The secret key $sk_t$ for time $t$ is given by:

$$\tilde{sk}_1, \tilde{sk}_{i+1}, \ldots, \tilde{sk}_T$$

generated using independent randomness. To update from the key $sk_t$ to $sk_{t+1}$, we simply erase $\tilde{sk}_t$. Forward security follows from the fact that an adversary who corrupts a signer at time $t$ only learns $sk_t$ and, in particular, does not learn $sk_{t'}$ for $t' < t$, and is unable to create signatures for past time slots.

To compress the secret keys down to $O((\log T)^2)$ without increasing the signature size, we combine the tree-based approach in [23] with the compact HIBE in [15]. Roughly speaking, each $sk_t$ now contains $\log T$ sub-keys, each of which contains $O(\log T)$ group elements and looks like an “expanded” version of $\tilde{sk}_t$. (In the simplified scheme, each $sk_t$ contains $T - t + 1$ sub-keys, each of which contains three group elements.)

Security against rogue-key attacks. The design of multi-signature schemes must take into account rogue-key attacks, where an adversary forges a multi-signature by providing specially crafted public keys that are correlated with the public keys of the honest parties. We achieve security against rogue-key attacks by having users provide a proof of possession of their secret key [14, 56]; it suffices here for each user to provide a standard BLS signature $y'$ on its public key $y$ (cf. the proof $\pi$ in the key generation and verification algorithms in Section 5.2).

Avoiding trusted set-up. Note that the common parameters contain uniformly random group elements $h, h_0, \ldots, h_{\log T}$ in $G_2$ which are used to define the function $F$. These elements can be generated using an indifferentiable hash-to-curve algorithm [20, 63] evaluated on some fixed sequence of inputs (e.g. determined by the digits of $pi$), thereby avoiding any trusted set-up.

2.1 Discussion

Related works. The use of HIBE schemes for forward secrecy originates in the context of encryption [23] and has been used in signatures [19, 27], key exchange [35] and proxy re-encryption [32]. Our signature scheme is quite similar to the forward-secure signatures of Boyen et al. [19] and achieves the same asymptotic complexity; their construction is more complex in order to achieve security against untrusted updates. The way we achieve aggregation is similar to the multi-signatures in [45].

Alternative approaches to posterior security. There are two variants of the posterior attack: (i) a short-range variant, where an adversary tries to corrupt a committee member prior to completion of the consensus sub-protocol, and (ii) a long-range variant as explained earlier. Dfinity [36], Ouroboros [41] and Casper [22] cope with the short-range attacks by assuming a delay in attacks that is longer than the running time of the consensus sub-protocol. For long-range attacks, Casper adopts a fork choice rule to never revert a finalized block, and in addition, assumes that clients log on with sufficient regularity to gain a complete update-to-date view of the chain. We note that forward-secure signatures provide a clean solution against both attacks, without the need for fork choice rules or additional assumptions about the adversary and the clients.

Application to permissioned blockchains. Consensus protocols, such as PBFT, are also at the core of many permissioned blockchains (e.g. Hyperledger), where only approved parties may join the network. Our signature scheme can similarly be applied to this setting to achieve forward secrecy, reduce communication bandwidth, and produce compact block certificates.

3 Preliminaries

Let $G_1, G_2, G_{t}$ be multiplicative groups of prime order $q$ with a non-degenerate pairing function $e : G_1 \times G_2 \rightarrow G_{t}$. Let $g_1$ and $g_2$ be generators of $G_1$ and $G_2$, respectively.

In analogy with the weak bilinear Diffie-Hellman inversion problem $\ell$-wBDH$^*$ [15], which was originally defined for Type-1 pairings (i.e., symmetric pairings where we have $G_1 = G_2$), we define the following variant for Type-3 pairings denoted $\ell$-wBDH$^*_{G_1 \times G_2}$.

\[\text{Input: } A_1 = g_1^\alpha, A_2 = g_2^\gamma, \ldots, A_t = g_1^{\alpha'}, B_1 = g_2^\alpha, B_2 = g_2^{\alpha'}, \ldots, B_t = g_2^{\alpha'}, C_1 = g_1^\gamma, C_2 = g_2^\gamma, \text{ for } \alpha, \gamma \in \mathbb{Z}_q
\]

\[\text{Compute: } e(g_1, g_2)^{(\gamma \alpha^{t+1})}
\]

The advantage \text{Adv}$^*_{\ell$-wBDH$^*_{G_1 \times G_2}}(A)$ of an adversary $A$ is defined as its probability in solving this problem.

As shown in [15], the assumption holds in the generic bilinear group model, with a lower bound of $\Omega(\sqrt{q/\ell})$ (with a matching attack in [26]). Concretely, for the BLS12-381...
pairing-friendly curve with \( \ell = 32 \), the best attack has complexity roughly \( 2^{125} \).

Alternatively, our scheme could be proved secure under a variant of the above assumption where the adversary has to output \( g_1^{(a^{i+1}_t)} \) given as input \( A_1, \ldots, A_t, B_1, \ldots, B_t \) and given access to an oracle \( \psi : g_2^s \mapsto g_1^i \). Because of the \( \psi \) oracle, this assumption is incomparable to the \( \ell \)-wBDHI assumption described above.

4 Forward-Secure Signatures

We begin by describing a forward-secure signature scheme, and then extend the construction to a multi-signature scheme in Section 5.

4.1 Definition

We use the Bellare-Miner model [9] to define syntax and security of a forward-secure signature scheme. A forward-secure signature scheme \( \mathcal{FS} \) for a message space \( \mathcal{M} \) consists of the following algorithms:

**Setup:** \( pp \leftarrow \text{Setup}(T) \). All parties agree on the public parameters \( pp \). The setup algorithm mainly fixes the distribution of the parameters given the maximum number of time periods \( T \). The parameters may be generated by a trusted third party, through a distributed protocol, or set to “nothing-up-my-sleeve” numbers. The public parameters are taken to be an implicit input to all of the following algorithms.

**Key generation:** \( (pk, sk_1) \leftarrow \text{Kg} \). The signer runs the key generation algorithm on input the maximum number of time periods \( T \) to generate a public verification key \( pk \) and an initial secret signing key \( sk_1 \) for the first time period.

**Key update:** \( sk_{t+1} \leftarrow \text{Upd}(sk_t) \). The signer updates its secret key \( sk_t \) for time period \( t \) to \( sk_{t+1} \) for the next period using the key update algorithm. The scheme could also offer a “fast-forward” update algorithm \( sk_{t'} \leftarrow \text{Upd'}(sk_t, t') \) for any \( t' > t \) that is more efficient than repetitively applying \( \text{Upd} \).

**Signing:** \( \sigma \leftarrow \text{Sign}(sk_t, M) \). On input the current signing key \( sk_t \) and message \( M \in \mathcal{M} \), the signer uses this algorithm to compute a signature \( \sigma \).

**Verification.** \( b \leftarrow \text{Vf}(pk, t, M, \sigma) \). Anyone can verify a signature \( \sigma \) for on message \( M \) for time period \( t \) under public key \( pk \) by running the verification algorithm, which returns 1 to indicate that the signature is valid and 0 otherwise.

**Correctness.**

Correctness requires that for all messages \( M \in \mathcal{M} \) and for all time periods \( t \in [T] \) it holds that

\[
\Pr[\text{Vf}(pk, t, M, \text{Sign}(sk_t, M)) = 1] = 1
\]

where the coin tosses are over \( pp \leftarrow \text{Setup}(T) \), \( (pk, sk_1) \leftarrow \text{Kg} \), and \( sk_i \leftarrow \text{Upd}(sk_{i-1}) \) for \( i = 2, \ldots, t \).

Moreover, if the scheme has a fast-forward update algorithm, then the keys it produces must be distributed identically to those produced by repetitive application of the regular update algorithm. Meaning, for all \( t, t' \in [T] \) with \( t < t' \leq T \) and for all \( sk_t \) it holds that \( sk'_t \leftarrow \text{Upd'}(sk_t, t') \) follows the same distribution as \( sk_t \) produced as \( sk_i \leftarrow \text{Upd}(sk_{i-1}) \) for \( i = t + 1, \ldots, t' \).

**Security.**

Unforgeability under chosen-message attack for forward-secure signatures is defined through the following game. The experiment generates a fresh key pair \( (pk, sk_1) \) and hands the public key \( pk \) to the adversary \( \mathcal{A} \). The adversary is given access to the following oracles:

**Key update.** If the current time period \( t \) (initially set to \( t = 1 \)) is less than \( T \), then this oracle updates the key \( sk_t \) to \( sk_{t+1} \) and increases \( t \).

**Signing.** On input a message \( M \), this oracle runs the signing oracle with the current secret key \( sk_t \) and message \( M \), and returns the resulting signature \( \sigma \).

**Break in.** The experiment records the break-in time \( \bar{t} \leftarrow t \) and hands the current signing key \( sk_t \) to the adversary. This oracle can only be queried once, and after it has been queried, the adversary can make no further queries to the key update or signing oracles.

At the end of the game, the adversary outputs its forgery \( (t^*, M^*, \sigma^*) \). It wins the game if \( \sigma^* \) verifies correctly under \( pk \) for time period \( t^* \) and message \( M^* \), if it never queried the signing oracle on \( M^* \) during time period \( t^* \), and if it queried the break-in oracle, then it did so in a time period \( \bar{t} > t^* \).

We define \( \mathcal{A} \)'s advantage \( \text{Adv}_{\mathcal{FS}}^{\text{cma}}(\mathcal{A}) \) as its probability in winning the above game.

We also define a selective variant of the above notion, referred to as sfu-cma, where the adversary first has to commit to \( \bar{t}, t^*, \) and \( M^* \). More specifically, \( \mathcal{A} \) first outputs \( (\bar{t}, t^*, M^*) \), then receives the public key \( pk \), is allowed to make signature and key update queries until time period \( t = \bar{t} \) is reached, at which point it is given \( sk_t \) and outputs its forgery \( \sigma^* \).

4.2 Encoding time periods

Following [23], we associate time periods with all nodes of the tree according to a pre-order traversal. Prior tree-based forward-secure signatures [9, 48] associate time periods with the only leaf nodes; using all nodes allows us to reduce the amortized complexity of key updates from \( O(\log T) \) exponentiations to \( O(1) \) exponentiations.

Recall that a tree of depth \( \ell - 1 \) has \( 2^\ell - 1 \) nodes, which then correspond to time periods in \([2^\ell - 1]\). We will identify
The sets \( \bar{t} \) happens to be the smallest set of nodes that includes a prefix as \( \ell \) any integer \( \leq \) \( |\bar{t}| \) of length exactly \( \ell \) given by
\[
|\bar{t}| = 2 - \ell - (|t| - 1) = 1 + \sum_{i=1}^{l} (2^{\ell-i} - 1).
\]

For instance, for \( \ell = 3 \), this maps \( \bar{t} \in \{1, 2, 3, 4, 5, 6, 7\} \). The inverse of the bijection can be described as
\[
t(1) = 0 \quad \text{and} \quad t(1) = 1 + \sum_{i=1}^{l} (2^{\ell-i} - 1)\).
\]

Next, we associate any \( \bar{t} \) to \( \bar{t} \|1 \) if \( |\bar{t}| \leq \ell - 1 \) and \( \bar{t} \|2 \) if \( |\bar{t}| = \ell - 1 \)

where \( \bar{t} \) is the longest string such that \( \bar{t} \|1 \) is a prefix of \( t(1) \).

The bijection induces a natural precedence relation over \( \{1, 2\} \subseteq \ell - 1 \) where \( \bar{t} \preceq \bar{t}' \) iff either \( \bar{t} \) is a prefix of \( \bar{t}' \) or exists \( \bar{t} \) s.t. \( \bar{t} \|1 \) is a prefix of \( \bar{t} \) and \( \bar{t} \|2 \) is a prefix of \( \bar{t}' \).

We can also describe the association explicitly as a bijection between \( t = t_1 | t_2 | \ldots \in \{1, 2\} \subseteq \ell - 1 \) and \( t \in \{2^\ell - 1\} \) for any integer \( \ell \) given by
\[
t(t) = 1 + \sum_{i=1}^{l} (2^{\ell-i} - 1)\).
\]

The first property is used for verification and for reasoning about security; the second and third properties are used for key updates.

### 4.3 Construction

We assume the bound \( T \) is of the form \( 2^\ell - 1 \). We use the above bijection so that the algorithms take input \( t \in \{1, 2\} \subseteq \ell - 1 \) instead of \( t \in [T] \). The following scheme is roughly the result of applying the Canetti-Halevi-Katz technique to obtain forward security from hierarchical identity-based encryption (HIBE) [24] to the signature scheme determined by the key structure of the Boneh-Boyen-Goh HIBE scheme [15]; we describe the differences at the end of this subsection.

**Setup.** Let \( M \) be the message space of the scheme and let \( H_q : M \rightarrow \{0, 1\}^* \) be a hash function that maps messages to bit strings of length \( k \) such that \( 2^k < q \). Apart from the description of the groups, the common system parameters also contain the maximum number of time slots \( T = 2^\ell - 1 \) and random group elements \( h, h_0, \ldots, h_t \in \mathbb{G}_1 \). These parameters could, for example, be generated as the output of a hash function modeled as a random oracle.

**Key generation.** Each signer chooses \( g \in \mathbb{Z}_q \) and computes \( y = g^x \). It sets its public \( pk = y \) and computes its initial secret key \( sk_1 = \{sk_t\} \) where \( sk_t = (g^x, h^x, h_0^x, \ldots, h_t^x) \) for \( r \in \mathbb{Z}_q \).

**Key update.** We associate with each \( w \in \{1, 2\}^k \) with \( k \leq \ell - 1 \) a key \( \bar{sk}_w \) of the form
\[
\bar{sk}_w = (\bar{c} \cdot \bar{d}, e_{k+1}, \ldots, e_\ell)
\]
for \( r \in \mathbb{Z}_q \). Given \( \bar{sk}_w \), one can derive a key for any \( w' \in \{1, 2\}^k \) which contains \( w \) as a prefix as
\[
(\bar{c}', \bar{d}', e_{k+1}', \ldots, e_\ell) = \left( c \cdot g_2^x \cdot \prod_{j=k+1}^{\ell} e_j^x \cdot (h_0 \prod_{j=1}^{\ell} h_j^x)^r \right) \cdot \left( e_{k+1} \cdot h_{k+1}^x, \ldots, e_\ell \cdot h_\ell^x \right)
\]
for \( r \in \mathbb{Z}_q \).

The secret key \( sk_t \) at time period \( t \) is given by
\[
sk_t = \{sk_w : w \in \Gamma_t\}
\]
which, by the first property of \( \Gamma_t \), contains a key \( \bar{sk}_w \) for a prefix \( w \) of all nodes \( t' \preceq t \).
To perform a regular update of \( sk_t \) to \( sk_{t+1} \), the signer users the second property of \( \Gamma_t \). Namely, if \( |t| < \ell - 1 \), then the signer looks up \( \bar{sk}_t = (c, d, e_{[t+1]}, \ldots, e_t) \in sk_t \), computes

\[
\bar{sk}_{t+1} \leftarrow (c, d, \cdot e_{[t+1]}, e_{[t+2]}, \ldots, e_t),
\]

and derives \( \bar{sk}_{t+2} \) from \( \bar{sk}_t \) using Equation (2). The signer then sets \( sk_{t+1} \leftarrow (sk_t \setminus \bar{sk}_t) \cup \{ \bar{sk}_{t+1}, \bar{sk}_{t+2} \} \) and securely deletes \( sk_t \) as well as the re-randomization exponent \( r' \) used in the derivation of \( \bar{sk}_{t+2} \).

If \( |t| = \ell - 1 \), then the signer simply sets \( sk_{t+1} \leftarrow sk_t \setminus \bar{sk}_t \) and securely deletes \( sk_t \).

To perform a fast-forward update of its key to any time \( t' \geq t \), the signer derives keys \( \bar{sk}_w' \) for all nodes \( w' \in \Gamma_t \setminus \Gamma_t' \) by applying Equation (2) to the key \( \bar{sk}_w \in sk_t \) such that \( w \in G_2 \). The signer then sets \( sk_t \leftarrow \{ \bar{sk}_w' : w' \in \Gamma_t' \} \) and securely deletes \( sk_t \) as well as all re-randomization exponents used in the key derivations.

**Signing.** To generate a signature on message \( M \in \mathcal{M} \) in time period \( t \in \{1, 2\}^{\leq \ell - 1} \), the signer looks up \( sk_t = (c, d, e_{[t]}, \ldots, e_t) \in sk_t \), chooses \( r' \in Z_q \), and outputs

\[
(\sigma_1, \sigma_2) = \left( d \cdot e_{H_t^c(M)}, (h_0 \cdot \prod_{j=1}^{\ell} h_{j^c} \cdot h_{t^c}^{H_t^c(M)} r', c \cdot g_2^r \right).
\]

**Verification.** Anyone can verify a signature \( (\sigma_1, \sigma_2) \in G_1 \times G_2 \) on message \( M \) under public key \( pk = y \) in time period \( t \) by checking whether

\[
ev(\sigma_1, \sigma_2) = e(h, y) \cdot e(h_0 \cdot \prod_{j=1}^{\ell} h_{j^c}^{h_{t^c}^{H_t^c(M)}}, c \cdot g_2^r).
\]

Note that the pairing \( e(h, y) \) can be pre-computed from the public key ahead of time, so that verification only requires two pairing computations.

**Differences from prior works.** We highlight the differences between our scheme and those in [15, 19, 23], assuming some familiarity with these prior constructions.

- We rely on asymmetric bilinear groups for efficiency, and our signature sits in \( G_2 \times G_1 \) instead of \( G_2^2 \). This way, it is sufficient to give out the public parameters \( h_0, \ldots, h_{\ell} \) in \( G_1 \) (which we can then instantiate using hash-to-curve without trusted set-up) instead of having to generate “consistent” public parameters \( (h_i, h_i^c) = (g_1^i, g_2^i) \in G_1 \times G_2 \).

- Our key-generation algorithm also deviates from that in the Boneh-Boyen-Goh HIBE, which would set

\[
pk = e(g_1, g_2)^c, h = g_1, sk_e = (g_2^r, g_1^r h_0^c, h_1^c, \ldots, h_\ell^c).
\]

In our scheme, \( pk = g_1^\ell \) lies in \( G_1^2 \) instead of \( G_1 \) and is therefore smaller. Setting \( h \) to be random instead of \( g_1 \) also allows us to achieve security under weaker assumptions. In fact, setting \( h = g_1 \) and \( pk = g_1^\ell \) would yield an insecure scheme in symmetric pairing groups where \( g_1 = g_2 \), since \( h^c = g_1^c = g_2^c = pk \).

### 4.4 Correctness

We say that a secret key \( sk_t \) for time period \( t \) is **well-formed** if \( sk_t = \{ sk_w : w \in \Gamma_t \} \), where each \( sk_w \) is of the form of Equation (1) for an independent uniformly distributed exponent \( r \in Z_q \). We first show that all honestly generated and updated secret keys are well-formed, and then proceed to the verification of signatures.

The key \( sk_t \) is trivially well-formed for \( t = 1 \), i.e., \( t = \varepsilon \), as can be seen from the key generation algorithm. We now show that \( sk_t \) is also well-formed after a regular update from time \( t \) to \( t + 1 \) and after a fast-forward update from \( t \) to \( t' \geq t \).

In a regular update, assume that \( sk_t \) is well-formed. If \( |t| = \ell - 1 \), then the update procedure sets \( sk_{t+1} \leftarrow sk_t \setminus \{ sk_1 \} \), which by the second property of \( \Gamma_t \) and the induction hypothesis means that \( sk_{t+1} \) is also well-formed. If \( |t| < \ell - 1 \), the update procedure adds keys \( sk_{t+1} \) and \( \bar{sk}_{t+2} \) and removes \( sk_t \) from \( sk_{t+1} \), which by the second property of \( \Gamma_t \) indeed corresponds to \( \{ w : w \in \Gamma_{t+1} \} \). Moreover, \( sk_{t+1} \) is derived from \( \bar{sk}_t = sk_{t+1} \leftarrow (c, d, e_{[t+1]}, \ldots, e_t) \) as \( sk_{t+1} \leftarrow (c, d \cdot e_{[t+1]}, e_{[t+2]}, \ldots, e_t) \), which satisfies Equation (1) with randomness \( r \) that is independent from all other keys in \( sk_{t+1} \) because \( sk_t \notin sk_{t+1} \). Similarly, \( \bar{sk}_{t+2} \) satisfies Equation (1) because it is generated as

\[
c' = c \cdot g_2^{r'} = g_2^{r+1},
\]

\[
d' = d \cdot e_{k+1} \cdot (h_0 \prod_{j=1}^{k} h_{j^c}^{h_{k+1}^{w_{j+1}}})^{r'}
\]

\[
h' = h_{k+2}^{r'},
\]

\[
e_{k+2} \cdot h_{k+2}^{r'} = h_{k+2}^{r'}.
\]

satisfying Equation (1) with randomness \( r + r' \), which is independent of the randomness of other keys in \( sk_{t+1} \) due to the uniform choice of \( r' \).

For the fast-forward update procedure, one can see that if \( sk_t \) is well-formed, then the updated key \( sk_{t'} \) for \( t' > t \) is well-formed as well. Indeed, by adding the keys for nodes in \( \Gamma_{t'} \setminus \Gamma_t \) and removing those for \( \Gamma_t \setminus \Gamma_{t'} \), we have that \( sk_{t'} \) contains keys \( sk_w \) for all \( w \in \Gamma_{t'} \). The randomness independence is guaranteed by the random choice of \( r' \) in Equation (2). In the
Theorem 1. For any fu-cma adversary $A$ against the above forward-secure signature scheme in the random-oracle model for $T = 2^l - 1$ time periods, there exists an adversary $B$ with essentially the same running time and advantage in solving the $\ell$-wBDHI$_3$ problem

$$\text{Adv}^{\text{wBDHI}_3}_{G \times G_2}(B) \geq \frac{1}{T \cdot q_H} \cdot \text{Adv}^{\text{fu-cma}}_{f_S}(A) - \frac{q_H^2}{2^\ell},$$

where $q_H$ is the number of random-oracle queries made by $A$.

We refer the interested reader to Appendix A for the full proof of security.

5 Forward-Secure Multi-Signatures

To obtain a multi-signature scheme, we observe that the component-wise product $(\Sigma_1, \Sigma_2) = (\prod_{i=1}^n \sigma_{i1}, \prod_{i=1}^n \sigma_{i2})$ of a number of signatures $(\sigma_{11}, \sigma_{12}), \ldots, (\sigma_{n1}, \sigma_{n2})$ satisfies the verification equation with respect of the product of public keys $Y = y_1 \cdot \ldots \cdot y_n$. This method of combining signatures is vulnerable to a rogue-key attack, however, where a malicious signer chooses his public key based on that of an honest signer, so that the malicious signer can compute valid signatures for their aggregated public key. The scheme below borrows a technique due to Ristenpart and Yilek [56] using proofs of possession (denote by $\pi$ below) to prevent against these types of attack.

5.1 Definitions

In addition to the algorithms of a forward-secure signature scheme in Section 4.1, a forward-secure multi-signature scheme $F_{\text{MAES}}$ in the key verification model has a key generation that additionally outputs a proof $\pi$ for the public key:

**Key generation:** $(pk, \pi, sk_1) \leftarrow \text{KAgg}$. The key generation algorithm generates a public verification key $pk$, a proof $\pi$, and an initial secret signing key $sk_1$ for the first time period. and additionally has the following algorithms:

**Key verification:** $b \leftarrow \text{KVf}(pk, \pi)$. The key verification algorithm returns 1 if the proof $pk$ is valid for $pk$ and returns 0 otherwise.

**Key aggregation:** $(pk, \pi) \leftarrow \text{Aaggg}(pk_1, \ldots, pk_n)$. On input a list of individual public keys $(pk_1, \ldots, pk_n)$, the key aggregation returns an aggregate public key $apk$, or $\perp$ to indicate that key aggregation failed.

**Signature aggregation:** $\Sigma \leftarrow \text{SAgg}(sk_1, \pi_1, \ldots, (pk_n, \sigma_n), t, M)$. Anyone can aggregate a given list of individual signatures $(\sigma_1, \ldots, \sigma_n)$ by different signers with public keys $(pk_1, \ldots, pk_n)$ on the same message $M$ and for the same period $t$ into a single multi-signature $\Sigma$.

**Aggregate verification:** $b \leftarrow \text{AVf}(apk, t, M, \Sigma)$. Given an aggregate public key $apk$, a message $M$, a time period $t$, and a multi-signature $\Sigma$, the verification algorithm returns 1 to indicate that all signers in $apk$ signed $M$ in period $t$, or 0 to indicate that verification failed.

**Correctness.** Correctness requires that $\text{KVf}(pk, \pi) = 1$ with probability one if $(pk, \pi, sk_1) \leftarrow \text{KAgg}$ and that for all messages $M \in \mathcal{M}$, for all $n \in \mathbb{Z}$, and for all time periods $t \in \{0, \ldots, T-1\}$, it holds that $\text{AVf}(apk, t, M, \Sigma) = 1$ with probability one if $(pk, \pi, sk_1) \leftarrow \text{KAgg}(pk_1, \ldots, pk_n)$, $sk_{i,j} \leftarrow \text{Upd}(sk_{i,j-1})$ for $i = 1, \ldots, n$ and $j = 2, \ldots, t$, $\Sigma \leftarrow \text{SAgg}(sk_1, \pi_1, \ldots, (pk_n, \sigma_n), t, M)$.

**Security.** Unforgeability (fu-cma) is defined through a game that is similar to that described in Section 4.1. The adversary is given the public key $pk$ and proof $\pi$ of an honest signer and access to the same key update, signing, and break-in oracles. However, at the end of the game, the adversary’s forgery consists of a list of public keys and proofs $(pk_1^*, \pi_1^*, \ldots, pk_n^*, \pi_n^*)$, a message $M^*$, a time period $t^*$, and a multi-signature $\Sigma^*$. The forgery is considered valid if

- $pk \in \{pk_1^*, \ldots, pk_n^*\}$,
- the proofs $\pi_1^*, \ldots, \pi_n^*$ are valid for public keys $pk_1^*, \ldots, pk_n^*$ according to $\text{KVf}$,
• Σ* is valid with respect to the aggregate public key apk* of \((p_k^0, \ldots, p_k^n)\), message \(M^*\), and time period \(t^*\).
• \(r > t^*\),
• and \(\mathcal{A}\) never made a signing query for \(M^*\) during time period \(t^*\).

Our security model covers rogue-key attacks because the adversary first receives the target public key \(p_k\) and only then outputs the list of public keys \(p_k^0, \ldots, p_k^n\) involved in its forgery. The only condition on these public keys is that they are accompanied by valid proofs \(\pi_1^*, \ldots, \pi_n^*\).

5.2 Construction

Let \(H_{G_1} : \{0,1\}^* \rightarrow \mathbb{G}_1^*\) be a hash function. The multi-signature scheme reuses the key update and signature algorithms from the scheme from Section 4.3, but uses different key generation and verification algorithms, and adds signature and key aggregation.

Key generation. Each signer chooses \(x \in \mathbb{Z}_p\) and computes \(y \leftarrow g_2^x\) and \(y' \leftarrow H_{G_1}(\text{P}, y)\), where \(\text{P}\) is a fixed string used as a prefix for domain separation. It sets its public key to \(pk = y\), the proof to \(\pi = y'\), and computes its initial secret key as \(sk_1 \leftarrow h^x\).

Key verification. Given a public key \(pk = y\) with proof \(\pi = y'\), the key verification algorithm validates the proof of possession by returning 1 if
\[
e(y', g_2) = e(H_{G_1}(\text{P}, y), y)
\]
and returning 0 otherwise.

Key aggregation. Given public keys \(pk_1 = y_1, \ldots, pk_n = y_n\), the key aggregation algorithm computes \(Y \leftarrow \prod_{i=1}^n y_i\) and returns the aggregate public key \(apk = Y\).

Signature aggregation. Given signatures \(\sigma_1 = (\Sigma_1, \Sigma_2)\), \(\sigma_n = (\Sigma_{n,1}, \Sigma_{n,2}) \in G_1 \times G_2\) on the same message \(M\), the signature aggregation algorithm outputs
\[
\Sigma = (\Sigma_1, \Sigma_2) = \left(\prod_{i=1}^n \Sigma_{i,1}, \prod_{i=1}^n \Sigma_{i,2}\right).
\]

Aggregate verification. Multi-signatures are verified with respect to aggregate public keys in exactly the same way as individual signatures with respect to individual public keys. Namely, given a multi-signature \((\Sigma_1, \Sigma_2) \in G_1 \times G_2\) on message \(M\) under aggregate public key \(apk = Y\) in time period \(t\), the verifier accepts if and only if \(apk \neq \perp\) and
\[
e(\Sigma_1, g_2) = e(h, Y) \cdot e(h_0 \prod_{j=1}^{|T|} h_j^j, h_{t+1}^{|T|}, \Sigma_2).
\]

5.3 Security

Theorem 2. For any fu-cma adversary \(A\) against the above forward-secure multi-signature scheme for \(T = 2^k - 1\) time periods in the random-oracle model, there exists an adversary \(B\) with essentially the same running time that solves the \(e\)-wBDHI problem with advantage
\[
\text{Adv}^{	ext{e-wBDHI}}_{G_1 \times G_2}(B) \geq \frac{1}{T \cdot q_H} \cdot \text{Adv}^{	ext{fu-cma}}_{\mathcal{F}_{\mathcal{A}}} (A) - \frac{q_H^2}{2^k},
\]
where \(q_H\) is the number of random-oracle queries made by \(A\).

We prove the theorem by showing that a forger \(\mathcal{A}\) for the multi-signature scheme yields a forger \(\mathcal{A}'\) for the single-signer scheme of Section 4.3 such that \(\text{Adv}^{	ext{fu-cma}}_{\mathcal{F}_{\mathcal{A}}} (A) \geq \text{Adv}^{	ext{fu-cma}}_{\mathcal{F}_{\mathcal{A}'}} (A)\). The theorem then follows from Theorem 1.

The key idea for the proof of security following [56] is to program \(H_{G_1}\) in such a way that we can “extract” a valid forgery for the single-signer scheme starting from that for the multi-signature scheme. In particular,

• given a rogue public key \(pk_i = y_i\) with proof \(\pi_i = y_i'\) where \(y_i = g_2^x\), we can extract the corresponding secret key \(h^y_i\) from \(y_i'\) by programming \(H_{G_1}(\text{P}, y_i) = h^y_i\).
• given \(h^y_i\) for all \(y_i \neq y\) along with a valid forgery for the multi-signature scheme, we can extract a forgery for the single-signer scheme. Here, we use the proofs \(\pi_i = (h_i^y)'\) to extract \(h_i^y\) for all adversarial keys \(pk_i = g_2^y\).

We defer the interested reader to Appendix B for proof details.

6 Pixel in PoS-based Blockchains

In this section, we describe how to integrate Pixel into PoS-based blockchains that rely on forward-secure signatures to achieve security against posterior corruptions. We summarize systems that rely on forward-secure signatures, abstract how signatures are used in these systems, and explain how to apply Pixel.

PoS Blockchains Secure under Posterior Corruptions.

Ouroboros Genesis and Praos rely on forward-secure signatures to protect against posterior corruptions [5, 31, 40]. These blockchains require users to rotate key and assume secure erasures. Thuderella is a blockchain with fast optimistic instant confirmation [54]. The blockchain is secure against posterior corruptions assuming that a majority of the computing power is controlled by honest players. Similarly, the protocol relies on forward-secure signatures. Pixel can be applied in all these blockchains to protect against posterior attacks and potentially reduce bandwidth, storage, and computation costs in instances where many users propagate many signatures on
the same message (e.g., a block of transactions). Ouroboros Cryptosinus uses forward-secure encryption to protect against the same attack [39]. Snow White shows that under a mild setup assumption, when nodes join the system they can access a set of online nodes the majority of whom are honest, the system can defend against posterior corruption attack [30]. The system does not rely on forward-secure signatures.

**Background on PoS Blockchains.** A blockchain is an append-only public ledger to which anyone can write and read. The fundamental problem in blockchains is to agree on a block of transactions between users. In Proof-of-Stake protocols, users map the stake or tokens they own in the system to “voting power” in the agreement protocol. Various types of PoS systems exist that use different formulas for determining the weight of each vote. For instance, in bounded PoS protocols, users must explicitly lock some amount $A$ of their tokens to participate in the agreement. The weight of each vote is $A/Q$, where $Q$ is the total number of locked tokens who’s users wish to participate in the agreement. Users that misbehave are punished by a penalty applied to their locked tokens.

To tolerate malicious users, all PoS protocols run a Byzantine sub-protocol to agree on a block of transactions. The system is secure, assuming that that majority (often 2/3) of the tokens participating in the consensus is honest. Each block is valid if a majority of committee members, weighted by their stake, approved it.

**Pixel Integration.** In order to vote on a block $B$, each member of the sub-protocol signs $B$ using Pixel with the current block number. The consensus is reached when we see a collection of $N$ committee member signatures $\sigma_1, \ldots, \sigma_N$ on the same block $B$, where $N$ is some fixed threshold. Finally, we will aggregate these $N$ signatures into a single multi-signature $\Sigma$, and the pair $(B, \Sigma)$ constitute a so-called block certificate and the block $B$ is appended to the blockchain.

**Registering public keys.** Each user who wishes to participate in consensus needs to register a participation signing key. A user first samples a Pixel key pair and generates a corresponding PoP. The user then issues a special transaction (signed under her spending key) registering the new participation key. The transaction includes PoP, PoS verifiers who are selected to run an agreement at round $r$, check (a) validity of the special transaction, and (b) validity of PoP. If both checks pass, the user’s account is updated with the new participation key. From this point, if selected, the user signs on blocks using Pixel.

**Voting generation.** To generate a vote on a block number $t$, users first update their keys to correspond to the round number. Subsequently, they sign the block using the correct secret key and propagate the signature to the network.

**Propagating and aggregating signatures.** Individual committee signatures will be propagated through the network until we see $N$ committee member signatures on the same block $B$. Note that Pixel supports non-interactive and incremental aggregation: the former means that signatures can be aggregated by any party after broadcast without communicating with the original signers, and the latter means that we can add a new signature to a multi-signature to obtain a new multi-signature. In practice, this means that propagating nodes can perform intermediate aggregation on any number of committee signatures and propagate the result, until the block certificate is formed. Alternatively, nodes can aggregate all signatures just before writing a block to the disk. That is, upon receiving enough certifying votes for a block, a node can aggregate $N$ committee members’ signatures into a multi-signature and then write the block and the certificate to the disk. To speed up verification of individual committee member signatures, a node could pre-compute $e(h, y)$ for the $y$’s corresponding to the users with the highest stakes.

**Key updates.** When using Pixel in block-chains, time corresponds to the block number or sub-steps in consensus protocols. Naively, when associating time with block numbers, this means that all eligible committee members should update their Pixel secret keys for each time a new block is formed and the round number is updated. Assume for simplicity that each committee member signs at most one block (if not, simply append a counter to the block number and use that as the time). If a user is selected to be on the committee at block number $t$, it should first update its key to $sk_t$ (Pixel supports “fast-forward” key updates from $sk_t$ to $sk_{t'}$ for any $t' > t$), and as soon as it signs a block, updates its key to $sk_{t+1}$ and then propagates the signature. In particular, there is no need for key updates when a user is not selected to be on the committee.

**Tweaking the scheme.** The blockchain stores Pixel public keys of all eligible committee members, as well as multi-signatures on each block. It is easy to see that we can tweak the Pixel scheme so that public keys live in the group $G_1$ (which has a more compact description) instead of $G_2$; this way, we can minimize the size of the blockchain as well as the cost of aggregate verification, which is dominated by the cost of multiplying $N$ public keys for large $N$. This change does come at a small cost since signing is performed over the slower $G_2$ instead of $G_1$. When instantiated with the BLS12-381 pairing-friendly curve, each public key is 48 bytes, and each multi-signature is $48 \times 96 = 144$ bytes independent of $N$. Moreover, we estimate signing to take less than 3 ms, and signature verification less than 5 ms for $T = 2^{30}$. More details are provided in Section C.
7 Evaluation on Algorand Blockchain

In order to measure the concrete efficiency gains of Pixel, we evaluate it on the Algorand blockchain [59,60].

Algorand Overview. Algorand is a Pure PoS (PPOS) system, where each token is mapped to a single vote in the consensus without any explicit bonding [59,60]. Some users may opt-out from participation, in which case their tokens are excluded from the total number of participating tokens (i.e., the denominator in the weight). Each user maintains an account on-chain that specifies her spending key, balance, consensus participation status, participation key, and other auxiliary information. A user wishing to perform a transaction must sign it with her corresponding secret key. Users run a Byzantine consensus algorithm to agree on a block of transactions following the high-level structure we outlined in the previous section. We call a block certificate to denote a collection of votes above a certain threshold approving a block. All users in the network validate and store block certificate (and the corresponding transactions) on disk. We refer to a node as a computer system running Algorand client software on the user’s behalf.

Verifier Vote Structure and Block Certificates. In Algorand, each valid vote for a block proposal includes (a) a proof that the verifier was indeed selected to participate in the consensus at round $r$, and (b) a signature on the block proposal. In more detail, each vote includes the following fields:

- Sender identifier which is represented by a unique public key registered on-chain (32 bytes).
- Round and sub-step identifiers (8 bytes).
- Block header proposal (32 bytes).
- A seed used as an input to a VRF function for cryptographic sortition (32 bytes).
- VRF credential that proves that the sender was indeed chosen to sign on the block (96 bytes).
- Forward-secure signature authenticating the vote (256 bytes).

Overall, each vote is about 500 bytes (including some additional auxiliary information), half of which is for the forward-secure signature.

Algorand has two voting sub-steps for each round. In the first sub-step, a supporting set (of expected size 3000) of verifiers is chosen to vote on a block proposal. In the second sub-step, a certifying set (of expected size 1500) of verifiers is chosen to finalize the block proposal. All verifiers’ votes propagate in the network during the agreement, but only the certifying votes are stored long-term and sufficient to validate a block in the future. Larger recovering set (of expected size 10000) is chosen during a network partition for recovery.

<table>
<thead>
<tr>
<th>Sig. set size</th>
<th>BM-Ed25519</th>
<th>BM-BLS</th>
<th>Pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256 B</td>
<td>192 Bytes</td>
<td>144 B</td>
</tr>
<tr>
<td>1500</td>
<td>375 KB</td>
<td>141 KB</td>
<td>144 B</td>
</tr>
<tr>
<td>3000</td>
<td>750 KB</td>
<td>281 KB</td>
<td>144 B</td>
</tr>
<tr>
<td>10000</td>
<td>2.4 MB</td>
<td>938 KB</td>
<td>144 B</td>
</tr>
</tbody>
</table>

Table 1: Total size of signature sets using various forward-secure signature schemes for $2^{32}$ time periods. BM-Ed25519 is instantiated using Algorand’s parameters with 10,000-ary tree of depth 2. BM-BLS is instantiated using the same parameters with public keys in $G_1$ and signatures in $G_2$.

Algorand’s Existing Solution to Posterior Corruptions. Algorand solves posterior corruptions using forward-secure signatures instantiated with a $d$-ary certificate tree [9], which we call BM-Ed25519 for convenience. The root public key of an Ed25519 signature scheme is registered on-chain, and keys associated with the leaves (and subsequently used to sign at each round) are stored locally by the potential verifiers. For each block at round $r$, a verifier must (a) produce a valid certificate chain from the root public key to the leaf associated with $r$, and (b) signature of the vote under the leaf key. Algorand assumes secure erasures and that users delete old keys from their nodes. BM-Ed25519 is instantiated with 10000-ary tree and depth 2 (supporting approximately $2^{26.6}$ time periods). Ed25519-based signatures have public keys of 32 bytes and 64 bytes signatures. Hence, since a valid certificate chain must include the intermediary public keys, the resulting size of each forward-secure signature is $3 \times 64 + 2 \times 32 = 256$ bytes.

7.1 Efficiency Evaluation

Pixel signatures can serve as a replacement of BM-Ed25519 in Algorand following the same design as outlined Section 6.

Setup. Our experiments are performed on a MacBook Pro, 3.5 GHz Intel Core i7 with 16 GB DDR3. We use Algorand’s open-source implementations of Pixel signatures, VRF functions, Ed25519 signing, and verification [1,2]. For blockchain applications, since the public key must live on-chain, we choose to place Pixel public keys in $G_1$, obtaining smaller public keys and faster key aggregation during verification. We set the maximum time epoch to $T = 2^{32} - 1$, which is sufficient to rotate a key every second for 136 years.

Figure 3 shows the runtime of individual Pixel algorithms, aggregation, and object sizes for the BLS12-381 curve [7]. Next, we measure quantities that affect all nodes participating in the system: the size of signature sets, bandwidth, and block verification time. In Pixel, the signature set corresponds to a single multi-signature.
keygen key sign aggregate verify agg. | aggregate verify agg. |
| | | (N = 1500) | (N = 3000) | pk | σ | sk |
| 1.03 ms | 1.8 ms | 2.8 ms | 7.2 ms | 13.9 ms | 8.3 ms | 48 B | 144 B | 43 kB |

Figure 3: Performance figures of the Pixel signature scheme algorithms, and the size of public keys, signatures, and secret keys when using a BLS12-381 curve. $N$ denotes the amount of signatures and keys aggregated, respectively. Maximum number of time periods is $T = 2^{32} - 1$.

**Storage Savings.** In Table 1, we compare the sizes of signature sets that are propagated (for supporting and verifying votes) and stored (for verifying votes) by all participating nodes. We instantiate BM-Ed25519 with Algorand parameters of 10000-ary and depth 2. For BM-BLS we place the public key in $G_1$ and signatures in $G_2$. Since BLS supports aggregation of signatures, we can compress all signatures in a certificate chain and the signature of the block into 96 B (note that the public key in the certificate chain cannot be compressed and adds an additional 96 B per signature). Furthermore, we can compress all signatures across votes. Pixel signatures authenticating a block with 1500 signatures are 2667x and 1003x times smaller than signature sets using BM-Ed25519 and BM-BLS, respectively.

In Figure 4, we show long-term blockchain storage improvements using Pixel signatures. We evaluate storage assuming various number of transactions in each block. Each transaction in Algorand is about 232 bytes. We also assume that the entire expected number of certifying verifiers (1500) are selected for each block. Given today’s block confirmation time of just under 4.3 seconds per block, Algorand blockchain should produce $10^6$ blocks every $\approx 50$ days and $10^8$ blocks every $\approx 13$ years. Pixel signatures improve blockchain size by about 40% and 20% on blocks packed with 1500 and 5000 transactions, respectively. This improvement translates to smaller overall storage requirements and faster catch-up speed for new nodes.

We clarify that the savings we obtain from Pixel are complimentary to those of Vault [44], which is another system built on top of Algorand to improve storage and catch-up speed. In particular, Vault can be used in conjunction with Pixel to obtain further storage savings. Vault creates “jumps” between blocks so that the system can confirm block $r$ knowing only block $r - k$ for some parameter $k$ (e.g., $k = 100$). Instead of downloading every block, a catch-up node in Vault only needs to download every $k$th block. Even using Vault, users would need to download and store about $10^6$ blocks for every $\approx 13$ years of blockchain operation.

**Bandwidth Savings.** Algorand uses a relay-based propagation model where users’ nodes connect to a network of relays (nodes with more resources). Without aggregation during propagation, Pixel savings for the bandwidth for both relays

<table>
<thead>
<tr>
<th>Number of connections</th>
<th>BM-Ed25519</th>
<th>Pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.4 MB</td>
<td>2.5 MB</td>
</tr>
<tr>
<td>10</td>
<td>11 MB</td>
<td>6.2 MB</td>
</tr>
<tr>
<td>100</td>
<td>109.9 MB</td>
<td>61.8 MB</td>
</tr>
</tbody>
</table>

Figure 4: Size of blockchain measured for different total number of blocks. The top two plots assume average of 1500 transactions per block and the bottom plots assume 5000 transactions per block. All plots assume average of 1500 certifying votes per block.

Table 2: Total bandwidth to propagate a set of 4500 signatures during consensus to agree on a block of transactions.
Table 3: Total runtime to verify signature sets authenticating a vote VRF credentials, and (c) time to verify transactions. In vote propagates once on each connection. In Table 2, we summarize savings for 4500 votes propagated during consensus for each block. From the table, we see that a relay with 10 connections saves about 44% of bandwidth. Bandwidth can be improved even further if Algorand relays were to aggregate multiple votes before propagating them to the users.

Block Verification Time Savings. Since verifying a Pixel multi-signature requires only 3 pairings in addition to multiplying all the public keys in the signature set, they are faster to verify than BM-Ed25519 signatures sets. Table 3 shows that a set of 3000 signatures can be verified about 65x faster. In Figure 5, we measure the overall savings on block verification time. Block verification time is broken into three main intervals: (a) time to verify vote signatures, (b) time to verify VRF credentials, and (c) time to verify transactions. In each interval, signature verification dramatically exceeds the time of any additional checks (e.g., check that the transaction amount is higher than the user’s balance). Blocks with 1500 and 5000 transactions can be verified 38% and 29% faster, respectively.

8 Variants and Extensions

Deterministic signatures. In practice, it is helpful to implement deterministic signing and key updates in order to protect against attacks arising from bad randomness. We can achieve using the standard technique [13] of deriving randomness from a random oracle.

More precisely, we assume a random oracle $H'$ that maps to $Z_q$, and when signing $M$ at time $t$, we use $r \leftarrow H'(r \text{rand-sign}, \tilde{s}_d, M, t)$. When updating the key from time $t$ to $t + 1$, we may need to compute $\tilde{s}_d t \| 1$ and $\tilde{s}_d t \| 2$ from $\tilde{s}_d$. The required randomness $r$ can be computed with $H(\text{rand-update}, \tilde{s}_d)$. Alternatively, if we wish to avoid additional use of a random oracle, we can rely on prior “forward-secure PRG techniques” [43].

Using internal nodes. For ease of exposition, our scheme only assigns time periods to leaf nodes in the tree. Alternatively, one could follow the approach of [24] to use all nodes in the tree, in a pre-order traversal, as time periods, which will improve the efficiency of the key update algorithm. Time periods are then identified by bit strings of length at most $\ell$, rather than exactly $\ell$ bits, and a signature in time period $t = t_1 \ldots t_d$ is a tuple of the form

$$(\sigma_1, \sigma_2) = \left( h^r \cdot \left( h_{0} \prod_{j=1}^{d} h_{j}^{h_{j}(M)} \right)^{r}, g_2^{r} \right).$$

Details are left to the reader.

Non-binary trees. One could try to reduce the key size by using $b$-ary trees instead of binary trees. A larger value of $b$ reduces the depth of the tree, but increases the amount of key material that must be kept at each level of the tree. To support $T$ time periods, one needs a $b$-ary tree of depth $\ell = \lfloor \log_b T \rfloor$. A node key at level $d$, however, can now take up to $b - 1$ keys of one element in $G_2$ and $(\ell + d - 2)$ elements of $G_1$. 

<table>
<thead>
<tr>
<th>Sig. set size</th>
<th>BM-Ed25519</th>
<th>Pixel</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.18 ms</td>
<td>4.9 ms</td>
<td>27x slower</td>
</tr>
<tr>
<td>1500</td>
<td>270 ms</td>
<td>6.7 ms</td>
<td>40x faster</td>
</tr>
<tr>
<td>3000</td>
<td>540 ms</td>
<td>8.3 ms</td>
<td>65x faster</td>
</tr>
<tr>
<td>10000</td>
<td>1.8 sec</td>
<td>15.6 ms</td>
<td>115x faster</td>
</tr>
</tbody>
</table>
The savings effect is quite limited, however, because the disadvantage of needing more keys per level quickly starts dominating the advantage of having less levels. For practical values of $T$, the maximum size of the secret key will usually be minimal for $b = 3$.

Parallel key timelines. In some applications, a signer may want to maintain several parallel timelines for different usages of a signing key. For example, in a sharded blockchain, the shards may be running in parallel at different speeds, without strict synchronization between the shards. If a time frame of the forward-secure signature scheme corresponds to the block height of a blockchain, for example, then the signer needs to maintain a different key schedule for the different shards.

A trivial approach is to run a separate instance of Pixel per timeline, and certify each public key with one root signing key. A more efficient approach for our particular scheme is to replace the fixed common parameter $h$ by the adversary $A$, by hashing the message into $G_1$ instead of into $\mathbb{Z}_q$. Namely, a multi-signature would be a tuple $(\Sigma_1, \Sigma_2, \Sigma_3)$ satisfying

$$e(\Sigma_1, g_2) = e(h, Y) \cdot e(h_{1,i} \prod_{j=1}^{\ell} h_{1,j}, \Sigma_2) \cdot e(H_{G_1}(\text{msg}, M), \Sigma_3).$$

This scheme has the additional advantage of saving up to $\ell$ elements of $G_1$ in secret key size, but signatures are one element of $G_2$ longer than the base scheme. We leave details to the reader.

Avoiding proofs-of-possession. In situations where proofs-of-possession are not desirable, one could alternatively reuse techniques from [16, 49] to avoid rogue-key attacks. Signers’ public keys are simply given by $pk_i = y_i = g_2^{x_i}$, but the aggregate public key is computed as $apk \leftarrow \prod_{i=1}^{n} pk_i^{h_i((pk_1, \ldots, pk_n, pk_i)}$. Individual signatures $(\sigma_{1,1}, \sigma_{1,2}), \ldots, (\sigma_{n,1}, \sigma_{n,2})$ are aggregated as

$$(\Sigma_1, \Sigma_2) \leftarrow (\prod_{i=1}^{n} \sigma_{1,i}^{h_i((pk_1, \ldots, pk_n, pk_i)} \prod_{i=1}^{n} \sigma_{2,i}^{h_i((pk_1, \ldots, pk_n, pk_i)}),$$

so that verification can be performed as usual.

9 Conclusion

In this work, we focus on improving the speed and security of PoS consensus mechanisms via optimizing its core building block – digital signature scheme. We design a new pairing-based forward-secure multi-signature scheme, Pixel. We prove that Pixel is secure in the random oracle model under a variant of Diffie-Hellman inversion problem over bilinear groups. Pixel is efficient as a stand-alone primitive and results in significant performance and size reduction compared to the previous forward-secure signatures applied in settings where multiple users sign the same message (block). For instance, compared to a set of 1500 tree-based forward-secure signatures, a single Pixel signature that can authenticate the entire set is 2667x smaller and can be verified 40x faster. We explained how to integrate Pixel to any PoS blockchains to solve posterior corruptions problem. We also demonstrate that Pixel provides significant efficiency gains when applied to Algorand blockchain. Pixel signatures reduce the size of Algorand blocks with 1500 transactions by $\approx 35\%$ and reduce block verification time by $\approx 38\%$.

Acknowledgments

We would like to thank Zhenfei Zhang for implementing Pixel as well as his help with Section 7. In addition, we thank Jens Groth, Nickolai Zeldovich, our shepherd Ari Juels, and the anonymous reviewers for useful feedback.

References


### A Security Proof of Forward-Secure Signatures

**Proof.** We prove the theorem in two steps. First, we show that the scheme is selectively secure when the message space $\mathcal{M} = \{0,1\}^k$ and $H_q$ is the identity function, meaning, interpreting a $k$-bit string as an integer in $Z_q$.

Step 1: sfu-cma. We show that the above scheme with message space $\mathcal{M} = \{0,1\}^k$ and $H_q$ the identity function is sfu-cma-secure under the $\ell$-wBDHI assumption by describing an algorithm $A$ that, given a successful sfu-cma forger $A'$, solves the $\ell$-wBDHI problem. On input $(A_1 = g_1^{\alpha_1}, A_2 = g_1^{\alpha_2}, \ldots, A_{\ell} = g_1^{\alpha_{\ell}}, B_1 = g_2^{\alpha_1}, \ldots, B_{\ell} = g_2^{\alpha_{\ell}}, C)$, algorithm $A$ proceeds as follows.

1. First, $A$ decides $\alpha_{\ell}$. That is, $A$ receives $s_{\ell}$ and produces a forgery on $t^*, \mathcal{M}^*$. Let $w^* \in \{0,1\}^{\ell-1}$ such that $w^* = w_1^{\alpha_1} \cdots w_{\ell-1}^{\alpha_{\ell-1}} = t^* [0^{\ell-1} - t^*]$. It then sets the public
key and public parameters as
\[
\begin{align*}
y & \leftarrow B_1 \\
h & \leftarrow g_1^y \cdot A_t \\
h_0 & \leftarrow g_1^y \cdot \prod_{i=1}^{t-1} A_{t-i+1}^{-w_i} \cdot A_1^{-M'} \\
h_{i} & \leftarrow g_1^y \cdot A_{t-i+1}^{-w_i} \quad \text{for } i = 1, \ldots, \ell ,
\end{align*}
\]
where \(\gamma, \gamma_0, \ldots, \gamma_{\ell} \leftarrow Z_q\).

By setting the parameters as such, \(B\) implicitly sets \(x = \alpha\) and \(h^x = A_1^y \cdot g_1^{(e^{l+1})}\). The reduction allows us to achieve two goals:

- extract the value of \(h^x\) from a forgery on \(t, M\) (provided by \(A_t\)), allowing \(B\) to easily compute its \(\ell\)-wBDHI solution \(e(g_1, C)(\alpha^{e^{l+1}})\);
- simulate \(s_k w\) for all \(w \in \{0, 1, 2\}^{\leq \ell-1}\) which are not a prefix of \(\gamma\); this would be useful for simulating both the signing and the break-in oracle.

Algorithm \(B\) responds to \(A_t\)'s oracle queries as follows.

**Key update.** There is no need for \(B\) to simulate anything beyond keeping track of the current time period \(t\).

**Signing.** We first describe how to answer a signing query for a message \(M\) in time period \(t \neq t^*\), and then describe the case that \(t = t^*\) and \(M \neq M^*\). Let \(w \in \{0, 1, 2\}^{\ell-1}\) be such that \(w = t[0^{\ell-1}-]t\).

Case 1: \(t \neq t^*\). It is easy to see that \(t \neq t^* \Rightarrow w \neq \gamma^*\). (This crucially uses the fact that \(t, t^* \in \{1, 2, \ldots, \alpha\}^*\).) Then, let \(w^* = w_1 \cdots w_{\ell}\) denote the shortest prefix of \(w\) which is not a prefix of \(w^*\). Extending the notation of \(s_k w\) to \(w^* \in \{0, 1, 2\}^{\leq \ell-1}\), we describe how \(B\) can derive a valid key \(s_k w\), from which it is straightforward to derive both \(s_k w\) and a signature for \(t, M\). Recall that \(s_k w\) has the structure
\[
(c, d, e_{k+1}, \ldots, e_{\ell}) = \left( g_2^\gamma, h^x(h_0 \prod_{i=1}^{k} h_i^{w_i})^r, h_{k+1}^{r}, \ldots, h_{\ell}^{r} \right)
\]
for a uniformly distributed value of \(r\). Focusing on the second component \(d\) first, we have that
\[
d = h^x \cdot \left(h_0 \cdot \prod_{i=1}^{k} h_i^{w_i} \right)^r = \left( g_1^{\gamma} A_t \right)^{\alpha} \cdot \left( \prod_{i=1}^{k} A_{t-i+1}^{-w_i} \cdot A_1^{-M'} \right)^{\alpha} = A_{1}^{\gamma} (\alpha^{e^{l+1}}) \cdot \left( \prod_{i=1}^{k} A_{t-i+1}^{-w_i} \cdot A_1^{-M'} \right)^{r},
\]
where the third equality holds because \(w_i = w_i^*\) for \(1 \leq i < k\) and \(w_k \neq w_k^*\). (Note that in the product notation \(\prod_{i=1}^{k-1}\) above, we let the result of the product simply be the unity element if \(k + 1 \geq \ell - 1\).) Let us denote the four factors between parentheses in the last equation as \(F_1, F_2, F_3,\) and \(F_4\), and denote their product as \(F\). If we let
\[
r \leftarrow r' + \frac{\alpha^k}{w_k^\gamma - w_k} \mod q
\]
for a random \(r' \leftarrow Z_q\), then we have that
\[
d = A_1^\gamma \cdot g_1^{(e^{l+1})}, F', \cdot F^\frac{\alpha^k}{w_k^\gamma - w_k}.
\]
The first and third factors in this product are easy to compute. The second factor would allow \(B\) to compute the solution its \(\ell\)-wBDHI problem as \(e(g_1^{(e^{l+1})}, C)\), so \(B\) cannot simply compute it. The last factor \(\frac{\alpha^k}{w_k^\gamma - w_k}\) can be written as the product of
\[
\begin{align*}
F_1 &= A_k^{-w_k^\gamma}, \\
F_2 &= A_{t-k+1}^{-w_k^\gamma}, \\
F_3 &= \prod_{i=1}^{k} A_{t-i+1}^{-w_k^\gamma}, \\
F_4 &= A_{t-k+1}^{-w_k^\gamma},
\end{align*}
\]
Because \(1 \leq k \leq \ell - 1\), it is clear that all but the second of these can be computed from \(B\)'s inputs, and that the second cancels out with the factor \(g_1^{(e^{l+1})}\) in \(d\), so that it can indeed compute \(d\) this way. The other components of the key are also efficiently computable as
\[
c = g_2^\gamma \cdot B_k^{-1} \\
e_{i} = h_i^{r}, \quad A_{t-i+1}^{-w_i} \quad \text{for } i = k+1, \ldots, \ell \\
e_{i} = h_i^{r}, \quad A_{t-i}^{-w_i} \quad \text{for } i = 0, \ldots, \ell - k - 1.
\]
From this key \((c, d, e_{k+1}, \ldots, e_{\ell})\) for \(w^*\), \(B\) can derive a key for \(w\) and compute a signature as in the real signing algorithm.

Case 2: \(t = t^*, M \neq M^*\). For a signing query with \(t = t^*\) but \(M \neq M^*\), \(B\) proceeds in a similar way, but derives the signature \((\sigma_1, \sigma_2)\) directly. Algorithm \(B\) can generate a valid signature using a similar approach as above, but using the fact that \(M \neq M^*\) instead of \(w_k \neq w_k^*\). Namely, letting
Break in. Here, $B$ needs to simulate $sk_w$ for all $w \in \Gamma_t$. This in turn requires simulating $sk_w$ for all $w \in \Gamma_t$. By the first property of $\Gamma_t$ (described in Section 4.2), all of these $w$ are not prefixes of $t'$ and also not prefixes of $w^*$, and we can therefore simulate $sk_w$ exactly as before.

Forgery. When $A'$ outputs a forgery $(\sigma^*_1, \sigma^*_2)$ that satisfies the verification equation
\[ e(\sigma^*_1, g_2) = e(h, y) \cdot e(h_0, \prod_{i=1}^{|t|} h_i^{t_i} \cdot h_t^M, \sigma^*_2), \]
then there exists an $r \in \mathbb{Z}_q$ such that
\[ \sigma^*_1 = h^r \cdot (h_0, \prod_{i=1}^{\ell} h_i^{t_i} \cdot h_t^M)^r, \]
\[ \sigma^*_2 = g_2^r. \]
From the way that $B$ chose the parameters $h, h_0, \ldots, h_t$, one can see that
\[ \sigma^*_1 = A_1^r, g_1^{(\ell+1)}, (g_1^{\gamma})^{h_0 + \sum_{i=1}^{\ell} t_i h_i^t} \cdot h_t^M. \]
Note that we do not know $g_1^r$, so we cannot directly extract $g_1^{\ell+1}$ from $\sigma^*_1$. Instead, observe that we have
\[ e(\sigma^*_1, C_2) = e(A_1^r, C_2) \cdot e(g_1^{(\ell+1)}, C_2) \cdot e(C_1, \sigma^*_2)^{h_0 + \sum_{i=1}^{\ell} t_i h_i^t} \cdot h_t^M, \]
from which $B$ can easily compute its output $e(g_1^{(\ell+1)}, C_2) = e(g_1, g_2)^{r \cdot t^{\ell+1}}$. It does so whenever $A'$ is successful, so that
\[ \text{Adv}^{\text{wu-BDH}_1}_{G_1 \times G_2} (B) \geq \text{Adv}^{\text{fu-cma}}_{A'} (A'). \]

**Step 2:** fu-cma. Full fu-cma security for $M = \{0,1\}$ and with $H_q : \mathcal{M} \to \{0,1\}^k$ modeled as a random oracle then follows because, given an fu-cma adversary $A'$ in the random-oracle model, one can build a sfu-cma adversary $A'$ that guesses the time period $r'$ and the index of $A'$'s random-oracle query for $H_q(M^*)$, and sets $t' \leftarrow t' + 1$. If $A'$ correctly guesses $r'$, then it can use $sk_j$ to simulate $A'$'s signature, key update, and break-in queries after time $t'$ until $A'$'s choice of break-in time $t'$, at which point it can hand over $sk_j$.

If $A'$ moreover correctly guessed the index of $H_q(M')$, and if $A$ never made colliding queries $H_q(M) = H_q(M')$ for $M \neq M'$, then $A'$'s forgery is also a valid forgery for $A'$. Note that for $A'$ to be successful, it must hold that $t' > t^*$, so it must hold that $t' \geq t$. The advantage of $A'$ is given by
\[ \text{Adv}^{\text{fu-cma}}_{A'} (A') \geq \frac{1}{T} \cdot \frac{\text{Adv}^{\text{fu-cma}}_{A'} (A) - q_1}{2^r}. \quad (3) \]
where $q_1$ is an upper bound on $A'$'s number of random-oracle queries. Together with Equation (3), we obtain the inequality of the theorem statement.

**B Security Proof of Forward-Secure Multi-signatures**

**Proof.** We show how to construct a forgery $A$ for the multi-signature scheme yields a forgery $A'$ for the single-signer scheme of Section 4.3 such that
\[ \text{Adv}^{\text{fu-cma}}_{A'} (A') \geq \text{Adv}^{\text{fu-cma}}_{A} (A). \]
The theorem then follows from Theorem 1.

**Step 1:** simulating $A$'s view. On input the parameters $(T, h, h_0, \ldots, h_t)$ and a public key $y$ for the single-signer scheme, the single-signer forgery $A'$ chooses $r \leftarrow \mathbb{Z}_q^*$ and stores $(y, \bot, g_1^r)$ in a list $L$. It computes $y' \leftarrow y$ and runs $A$ on the same common parameters and target public key $pk = y$ and proof $\pi = y'$. Observe that $\pi$ is indeed a valid proof for $pk$ since $e(y', g_2) = e(H_{G_1}(\text{PoP}, y), y)$.

Algorithm $A'$ answers all of $A$'s key update, signing, and break-in oracle queries, as well as random-oracle queries for $H_q$, by simply relaying queries and responses to and from $A$'s own oracles. Queries to the random oracle for $H_{G_1}$ are answered as follows.

**Random oracle $H_{G_1}$.** On input $(\text{PoP}, z)$, $A'$ checks whether there already exists a tuple $(z, \cdot, v) \in L$. If so, it returns $v$. If not, it chooses $r \leftarrow \mathbb{Z}_q^*$, computes $v \leftarrow h^r$, adds a tuple $(z, r, v)$ to $L$ and returns $v \cdot y$.

**Step 2:** extracting a forgery. When $A$ outputs its forgery
\[ (pk_1, \pi_1^*, \ldots, pk_n, \pi_n^*), M^*, t^*, \Sigma^*, \]
algorithm $A'$ first verifies the proofs $\pi_1^*, \ldots, \pi_n^*$ for public keys $pk_1, \ldots, pk_n$ and computes the aggregate public key $apk^*$.
creating additional entries in $L$ if necessary. Let $pk_i^j = y_i = g_2^r_i$ and $\pi_i^j = y_i^j$. Looking ahead, if $pk_i^j$ passes key verification, then we have $y_i^j = (h^r_i)^n$ and since we know $r_i$, we will be able to “extract” $h^r_i \in G_1$.

If all keys are valid, then it holds that $y_i^j = H_{G_1}(PoP, y_i)^{r_i}$ for all $i = 1, \ldots, n$. Let $apk^* = Y$ be the aggregate public key. From the aggregate verification equation

$$e(\Sigma_1^*, g_2) = e(h, Y) \cdot e(h_0, \prod_{j=1}^{[\ell]} h_j^{y_i^j \cdot H_i^{t_i}(M_i^*)}, \Sigma_2^*)$$

and the fact that $Y = \prod_{i=1}^{n} y_i = Y \cdot g_2^\sum_{i=1}^{n} y_i^q$, we have that

$$e(\Sigma_1^*, g_2) = e(h, y) \cdot e(h, g_2)^\sum_{i=1}^{n} y_i^q. \cdot e(h_0, \prod_{j=1}^{[\ell]} h_j^{y_i^j \cdot H_i^{t_i}(M_i^*)}, \Sigma_2^*)$$

$$\Leftrightarrow e(\Sigma_1^*, g_2) = e(h, y). \cdot e(h_0, \prod_{j=1}^{[\ell]} h_j^{y_i^j \cdot H_i^{t_i}(M_i^*)}, \Sigma_2^*).$$

For all $y_i \neq y$, $A^\prime$ looks up the tuple $(y_i, r_i, v_i)$ in $L$. We know that $v_j = h^n_i$, and hence that $y_i^j = h^{n_i}$. By comparing the last equation above to the verification equation of the single-signer scheme, and by observing that $y_i^j = h^{n_i}$, we know that the pair

$$\sigma_i^1 \leftarrow \Sigma_1^* \cdot \prod_{i=1, y_i \neq y}^{n} y_i^{-1/r_i}$$

$$\sigma_i^2 \leftarrow \Sigma_2^*$$

is a valid forgery for the single-signer scheme, so $A^\prime$ can output $M^*, t^*, (\sigma_1^1, \sigma_1^2)$ as its forgery. □

### C Efficiency Analysis

We let $T = 2^\ell - 1$ denote the maximum number of time periods.

**Computational Efficiency.** The main operations are key generation, updating the key, signing, aggregating public keys, and verifying signatures.

- Key generation requires 1 exponentiation in each of $G_1$ and $G_2$.
- Key verification requires 2 pairings.
- Key update for an arbitrary number of time steps requires $\ell^2$ exponentiations and $2\ell^2$ multiplications in $G_2$ and $\ell$ exponentiations in $G_1$; key updates can of course be entirely precomputed, if necessary. Key updates from $t$ to $t + 1$ require $\ell - |t|$ exponentiation in $G_1$ and 1 exponentiation in $G_2$ (ignoring multiplications) if $|t| < \ell - 1$, and no group operations if $|t| = \ell - 1$. On average, this only requires

$$1/2 \cdot 0 + 1/4 \cdot 2 + 1/8 \cdot 3 + 1/16 \cdot 4 + \cdots \leq 1.5$$

exponentiations in $G_1$ and

$$1/2 \cdot 0 + 1/4 \cdot 1 + 1/8 \cdot 1 + 1/16 \cdot 1 + \cdots \leq 0.5$$

exponentiation in $G_2$. That is, irrespective of the maximum number of time periods $T$, the average work for updating the key does not exceed 1.5 and 0.5 exponentiations in $G_1$ and $G_2$, respectively.

- Signing requires 3 exponentiations and 4 multiplications in $G_1$ and 1 exponentiation in $G_2$. By precomputing

$$\sigma_{1,1} \leftarrow d \cdot (h_0 \prod_{j=1}^{[\ell]} h_j^{y_i^j})^{r_i}$$

$$\sigma_{1,2} \leftarrow e \cdot h_i^{r_i}$$

$$\sigma_2 \leftarrow c \cdot g_2^{r_i},$$

the signature can be computed as $\sigma_1 \leftarrow \sigma_{1,1} \cdot \sigma_{1,2}^{H_i(M)}$ once the message $M$ is known, bringing the online computation down to a single exponentiation.

- Aggregating $N$ public keys together costs $N - 1$ multiplications in $G_2$. Here, we ignore the cost of verifying proofs of possession, which should only be performed once per public key.

- Verification of a signature requires 3 pairings (or one 3-multi-pairing) and $\ell$ multiplications and 1 exponentiation in $G_1$, plus subgroup membership checks for $G_1$ and $G_2$.

**Space Efficiency.** We are mainly concerned with the size of the public parameters, public keys, secret keys, and signatures.

- The public parameters consist of $\ell + 2$ elements of $G_1$.
- Every public key is a single element of $G_2$.
- The size of $sk_i$ is $\ell(\ell - 1)/2$ elements in $G_1$ and $\ell$ elements in $G_2$.
- A signature consists of one element in $G_1$ and one element in $G_2$. 