

Threshold Signatures with Identifiable Abort from One-way Functions

Mojtaba Fadavi, David Jao, Samuel Jaques

Department of Combinatorics & Optimization, University of Waterloo,
200 University Ave. W, Waterloo, ON N2L 3G1, Canada
`{mfadavi,djao,sejaques}@uwaterloo.ca`

Abstract. A (T, N) -threshold signature scheme distributes a signing key among N participants so that any T participants can jointly generate a valid signature, while fewer than T cannot.

In this work, we first introduce a threshold one-time signature scheme with *identifiable abort* based on a variant of Lamport one-time signature and Shamir secret sharing, enabling authorized participants to jointly reconstruct signatures from partial signatures. We then extend the scheme to support multiple messages using multi-layer Merkle trees.

One-time signatures introduce specific coordination challenges: participants must agree on both the message and the one-time key index to avoid unsafe key reuse. We address this using a bulletin-board mechanism for consensus. Under this framework, our construction produces signatures with one message from the bulletin board to each signer and one response message from each signer, while the bulletin board may require additional communication to reach agreement. With a simple majority-vote bulletin board, the scheme signs in four rounds. Subject to this coordination constraint, the construction inherits the guarantees of the underlying threshold framework: Universal Composability (UC) unforgeability with identifiable abort, non-UC robustness, and a bulletin-board mechanism that is UC-secure with an honest super-majority and robust when the central aggregator is honest.

Keywords: Post-quantum Digital Signature, Threshold Signature Scheme, Universal Composability

Table of Contents

1	Introduction.....	3
1.1	Related Works	4
1.2	Our Contribution.....	4
2	Preliminaries	6
2.1	Tweaked One-Time Signature Schemes	7
2.2	Merkle Trees.....	8
2.3	Batch Signatures	9
2.4	Shamir Secret Sharing	10
3	Threshold Signature Scheme with Identifiable Abort.....	10
4	Security Model	13
4.1	Communication Model	14
4.2	Ideal Functionality	15
5	Signature With Bulletin Board	17
5.1	Security	23
6	Bulletin Board.....	23
6.1	Ideal Functionality	25
7	Robustness.....	25
8	Performance.....	27
A	Components.....	34
A.1	Tweakable Hash Functions.....	34
A.2	Winternitz one-time signature scheme.	35
A.3	Merkle Trees.....	36
B	Bulletin Board.....	37
B.1	Internal Functions	39
B.2	Helper Functions	41
C	Security Proof of Theorem 1	44
C.1	Bulletin Board	49

1 Introduction

Digital signatures are a fundamental primitive for authenticity, integrity, and non-repudiation in distributed systems. Applications are transitioning to *post-quantum* (PQ-secure) digital signatures, which rely on new hard assumptions such as lattice/code/isogeny-based assumptions [23, 26, 30, 45, 47], or use one-way functions [12, 18, 35, 43, 44, 46].

One-way functions are known to be not only sufficient but also necessary for constructing digital signatures [50], making hash-based signatures the most conservative PQ-secure construction. This minimality has a cost: lacking algebraic structure, hash-based schemes cannot reuse a single key across arbitrarily many messages the way lattice/code/isogeny-based schemes can, and are instead built from one-time or few-time primitives, which are bound to a public key using a Merkle tree and reveal some of their secret key material with each signature. This lack of algebraic structure makes constructing threshold signature schemes more challenging compared to lattice/code/isogeny-based schemes, and some variants such as blind signatures are known to admit no *black-box* construction from one-way functions at all [38].

Threshold cryptography distributes trust across multiple parties, enabling the secure collaborative generation of secrets, signatures, and ciphertexts. In a (T, N) -threshold signature scheme, any subset of at least $T \leq N$ participants can jointly produce a single signature under a common public key, while any coalition of fewer than T parties cannot forge a valid one. This decentralization removes single points of failure but introduces subtle security challenges, as protocols must handle adversaries ranging from *fully malicious* to *honest-but-curious* participants who follow the protocol yet try to infer extra information from their views and the transcript. A threshold scheme must satisfy basic notions of *correctness* (honest parties obtain valid signatures) and *unforgeability* (coalitions of fewer than T cannot forge a signature). Even with these properties, adversaries might deviate arbitrarily from the protocol and disrupt the intended behaviour; a scheme with *identifiable abort* (IA) gives a mechanism to identify parties who misbehave, and a stronger *robust* scheme ensures that honest parties will always output a valid signature, no matter what the adversarial players do [4, 25, 28, 33, 39, 51].

Minimal-round threshold signing reduces interaction but often weakens robustness and IA, reflecting a trade-off between communication and fault tolerance. Concretely, one-round (non-interactive) schemes have signers send a single share to an aggregator [42, 53]; two-round (semi-interactive) schemes typically follow a commit-then-sign structure [39]; and multi-round protocols use additional interaction to strengthen malicious security [25, 28, 37]. Schemes also differ by corruption model, from *static* adversaries that choose corruptions in advance [53] to stronger *adaptive* adversaries that corrupt parties during execution [16, 31, 42]. Finally, schemes differ in their *dealer assumptions*. With a *trusted dealer*, an external party samples the signing key and distributes the shares [53]. With a *verifiable dealer*, VSS (PVSS) lets parties (parties and outsiders) verify that the shares are consistent with a single secret [29, 48, 54]. In

the *dealerless* setting, parties instead run *Distributed Key Generation* (DKG), avoiding any single key holder and using broadcast/complaint mechanisms to ensure correctness [4, 25, 28, 39, 51].

1.1 Related Works

Classical (T, N) -threshold signatures are well established, achieving robustness, IA, and low round complexity [39, 42, 51, 53], while PQ-secure threshold signatures are less mature. Recent lattice/isogeny-based schemes can provide IA and robustness, but typically incur higher computation and communication costs [20, 24, 25, 28, 41]. Motivated by this gap, and by NIST’s call for multi-party and threshold schemes with explicit post-quantum considerations [10], we pursue a robust post-quantum low-round threshold signature scheme relying only on one-way functions.

A significant line of work has improved post-quantum threshold signatures based on lattices (e.g.: [1, 3, 7, 9, 19, 22, 25, 27, 32, 34, 37, 49]); notable recent works allow threshold ML-DSA [45] signatures [6, 17], robustness [28], or small round complexity [8]. Our work is essentially orthogonal to all of this, however, since our goal is a scheme relying only on one-way functions and no other hardness assumptions.

1.2 Our Contribution

Consider a simple and insecure threshold signature: all participants broadcast their shares of the secret key, and then any one of them reassembles the secret key and uses it to sign the message. Obviously, this is insecure because now the secret key is revealed and nefarious participants can use it to sign any message they want. To fix this, we would need a secret that can only sign one message. With certain one-time signatures, this is possible. Consider the Lamport one-time signature (LOTS) scheme [40]: its secret key consists of 2λ random values $(x_{i,b})_{i \in [\lambda], b \in \{0,1\}}$, and the public key is $(H(x_{i,b}))_{i \in [\lambda], b \in \{0,1\}}$, where H is a one-way function. To sign a λ -bit message $\text{msg} = (m_1, \dots, m_\lambda)$, the signer reveals x_{i,m_i} . To verify, one simply checks that each component of the signature hashes to the corresponding value in the public key.

This could form an extremely simple one-round, one-time threshold signature, but one can see that it is tremendously inefficient. Practical hash-based signature schemes like the NIST standard SLH-DSA [46] use Winternitz and FORS signatures, which are more efficient one-time or few-time signatures; however, the signature is a more complicated function of the secret key that involves several hash function evaluations, which are poorly suited for MPC.

To fix this, we use a more efficient variant of LOTS, called CFF-LOTS. Two complementary lines of work reduce LOTS’s overhead in different ways [11, 55]. The first [11] keeps only one secret value per bit position instead of two, compensating via a checksum appended to the digest that enforces a monotonicity condition on valid encodings. The second [55] instead requires that the index set

revealed by one message never contain the index set revealed by any other message; this property is captured by a 1-cover-free family (1-CFF) [36]. Concretely, rather than 2λ secret values $(x_{i,b})_{i \in [\lambda], b \in \{0,1\}}$, we choose e values $(x_i)_{i \in [e]}$ and an injective map $g : \{0,1\}^\lambda \rightarrow \binom{[e]}{\kappa}$, revealing only $(x_i)_{i \in g(\text{msg})}$ for a message msg ; since distinct messages map to distinct κ -subsets, some index in $g(\text{msg}') \setminus g(\text{msg})$ stays hidden after signing msg , preserving one-time unforgeability. Choosing parameters such that $\binom{e}{\kappa} \geq 2^\lambda$ allows e to be only slightly larger than λ with $\kappa \approx e/2$, reducing the public key from 2λ hashes to approximately λ hashes and the signature from λ preimages to approximately $\lambda/2$ preimages. We formalize CFF-LOTS and its threshold extension in the following sections.

To expand this into a multi-use signature, we simply bundle the public keys of many CFF-based LOTS into a Merkle tree and use the root as the overall public key. Unfortunately, this does not work exactly as described. One-time threshold signatures are vulnerable to what we call a “message mismatch” attack. Suppose an adversary knows $T - 1$ shares in a (T, N) -threshold scheme. The adversary can ask two different honest users to sign two different messages *with the same LOTS keypair*. The adversary will obtain one share each for the two signatures, and can use their own shares to complete the signatures. They have now obtained two signatures for the same public key, which is not considered secure.

Single-party one-time signatures avoid this problem by maintaining a state which tracks which secret keys have been used so that they can avoid signing the same thing twice. Other threshold signatures avoid this problem because multiple signatures pose no security risk. For us to avoid this attack, we need to ensure that honest users always agree on which message to sign before creating any shares of a signature. At its core, this is a consensus problem.

If we have access to a “bulletin board”, a mechanism that ensures consensus on the ordering of messages, participants can simply sign whichever messages appear on the bulletin board, using the ordering on the board to decide on which one-time keypairs to use. This is what we opt to do, but a bulletin board is a heavy mechanism. As far as we can tell, it is a necessity: if we require that honest parties always sign exactly the same message with the same one-time keypair, then any ordering on the keypairs means the signatures themselves provide a consensus on the ordering of messages. That is, secure thresholdizing a collection of one-time signatures implies a bulletin board (For example, [21] provide a bulletin board based on threshold signatures which functions almost as the inverse of our scheme: messages are added to the bulletin board once they are signed by the threshold scheme).

For this paper, we explicitly provide a simple bulletin board mechanism that is naturally compatible with our usage as consensus for one-time signatures: a designated aggregator maintains a “blockchain”, where each block contains the next message to sign and a hash of the previous block, and the aggregator requests that all users sign each block with their long-term secret key. Users are required to never sign two different blocks at the same position in the chain, so by declaring a block “valid” if and only if it has a sufficient number of signatures,

we ensure any two blocks must share an honest signer, and that signer enforces consistency.

Security Model. We prove unforgeability in the UC framework. Our ideal functionality provides an “opt-in” variant of identifiable abort, where if signature reconstruction fails, parties can provide proof that their contribution was valid. While an adversary could choose not to provide this proof (and thus perpetually avoid blame), this poses a limited threat to our scheme, as honest users can simply proceed without the adversary and produce a signature (see our description of robustness below).

We achieve this blame mechanism via *verification keys*. As described above, a LOTS signature reveals a subset of preimages corresponding to components of the public key. Thus, during key generation, each party’s shares of the secrets are hashed into a commitment which is distributed to all parties. A selective opening of this commitment, to only the secret components used for a particular signature, can be used to verify that the partial signature produced was valid. While this requires an optional extra round *if* the signing process fails, the disincentive of being blamed should deter such misbehaviour.

Our protocol is robust as well, although we do not prove this in the UC framework. Unlike schemes such as FROST [39] or Threshold Raccoon [25], our scheme is “asynchronous”: partial signatures do not require knowledge of the set of users who will produce the final signature. Hence, if party i and party j each contribute a partial signature, and later it is decided that party j provided a malformed value (or even if it is proven that party i provided a well-formed value), their contribution can be replaced by a partial signature from party k without requiring party i to re-sign. Hence, given a set of partial signatures, we only need to determine which ones came from honest parties to reproduce a valid signature, and our identifiable abort provides this. This means we do not need complicated schemes like ROAST [51] to track multiple sessions of users. Moreover, this means that if an adversarial player attempts to avoid blame by never producing a proof of validity, then honest users can simply ignore them and wait for other honest users to give valid contributions. In short, a misbehaving adversary can, at worst, add one round to the signing process.

Similarly, our bulletin board only needs a certain number of signatures to make progress; as long as the aggregator is honest, it should proceed as expected.

2 Preliminaries

In this section, we review the cryptographic primitives underlying our scheme. Background on tweakable hash functions is provided in Appendix A.1.

Notations. Let $\lambda \in \mathbb{N}$ be the security parameter, $[n] = \{1, \dots, n\}$, and $\mathbb{Z}_n = \{0, \dots, n-1\}$ be the set of integers modulo n . For a prime p , we denote by $\mathbb{F}_p$ the finite field with p elements. For a list L , we denote its n -th element by $L[n]$. If S is a set or a list, we write $|S|$ for its size and $x \xleftarrow{\$} S$ for sampling a

uniformly random element x from S . For an algorithm A , we write $x \leftarrow A(a)$ for the output x of A on input a . A function $\epsilon : \mathbb{N} \rightarrow \mathbb{R}^+$ is called *negligible* in λ if for every polynomial $p(x)$ and all sufficiently large λ , we have $\epsilon(\lambda) < \frac{1}{p(\lambda)}$. For $x \in \mathbb{N}$, we denote by $\log x$ the logarithm of x to base 2.

2.1 Tweaked One-Time Signature Schemes

Our scheme builds on LOTS and WOTS, both of which are one-time signature schemes. So, we give a general syntax covering both, built around a *public-key derivation* algorithm that recovers a candidate public key. In typical uses (e.g., [46]) such signatures have “tweaks” derived from their context, for domain separation, and we include that in the syntax here.

Definition 1 *A tweaked one-time signature scheme with parameters prm is a tuple*

$$\text{OTS} = (\text{OTS.KeyGen}, \text{OTS.Sign}, \text{OTS.DerivePK}, \text{OTS.Verify}),$$

where

- $(\text{sk}^{\text{O}}, \text{pk}^{\text{O}}) \leftarrow \text{OTS.KeyGen}(\text{prm}, 1^\lambda, t)$: Generates a one-time key pair $(\text{sk}^{\text{O}}, \text{pk}^{\text{O}})$ from a tweak t ;
- $\sigma^{\text{O}} \leftarrow \text{OTS.Sign}(\text{prm}, \text{sk}^{\text{O}}, \text{msg}, t)$: Generates a signature σ^{O} on message msg using a one-time secret key sk^{O} and tweak t
- $\text{pk}^{\text{O}'} \leftarrow \text{OTS.DerivePK}(\text{prm}, \text{msg}, \sigma^{\text{O}}, \text{pk}^{\text{cmpl}}, t)$: Derives a candidate public key $\text{pk}^{\text{O}'}$ from $(\text{msg}, \sigma^{\text{O}})$ and the complementary public-key information pk^{cmpl} , without access to sk^{O} . Here, pk^{cmpl} contains the components of pk^{O} that cannot be recovered from $(\text{msg}, \sigma^{\text{O}})$;
- $0/1 \leftarrow \text{OTS.Verify}(\text{prm}, \text{pk}^{\text{O}}, \text{msg}, \sigma^{\text{O}}, t)$: Outputs 1 iff σ^{O} is a valid signature on msg , for the given public key pk^{O} and tweak t .

The CFF-based LOTS is a variant of LOTS that reduces the signature size by mapping each message to a subset of secret-key indices, chosen from a 1-cover-free family [36].

Definition 2 (Cover-free families) *A κ -uniform 1-cover-free family $1\text{-CFF}(e, \lambda, \kappa)$ is a collection $\mathcal{B} = \{B_1, \dots, B_{2^\lambda}\} \subseteq 2^{[e]}$ of κ -element subsets with $B_i \setminus B_j \neq \emptyset$ for all $i \neq j$.*

Since all blocks have size κ , cover-freeness is equivalent to the blocks being pairwise distinct, so a $1\text{-CFF}(e, \lambda, \kappa)$ exists iff $2^\lambda \leq \binom{e}{\kappa}$.

CFF-based LOTS. Let $\lambda \in \mathbb{N}$ be a security parameter, $H : \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ be a tweakable hash function, $\text{prm} = (\text{prm}_{\text{LOTS}}, \text{prm}_{\text{msg}})$ be public parameters, and $\mathcal{B}$ a κ -uniform $1\text{-CFF}(e, \lambda, \kappa)$ with bijection $g_\lambda : \{0, 1\}^\lambda \rightarrow \mathcal{B}$. Then

$$\text{LOTS} = (\text{LKeyGen}, \text{LSign}, \text{LDerivePK}, \text{LVerify}),$$

an instance of Definition 1, is defined as:

- $(\text{sk}^\text{L}, \text{pk}^\text{L}) \leftarrow \text{LKeyGen}(\text{prm}, 1^\lambda, t)$: Samples $x_k \xleftarrow{\$} \{0, 1\}^\lambda$ for each $k \in [e]$, and sets $\text{sk}^\text{L} = (x_1, \dots, x_e)$, $\text{pk}^\text{L} = (y_1, \dots, y_e)$ with $y_k = \text{H}(\text{prm}_{\text{LOTS}}, t \| k, x_k)$.
- $\sigma^\text{L} \leftarrow \text{LSign}(\text{prm}, \text{sk}^\text{L}, \text{msg}, t)$: Computes $B_{\text{msg}} = g_\lambda(\text{H}(\text{prm}_{\text{msg}}, t, \text{msg})) \subseteq [e]$ and outputs $\sigma^\text{L} = (x_k)_{k \in B_{\text{msg}}}$.
- $\text{pk}^{\text{L}' } \leftarrow \text{LDerivePK}(\text{prm}, \text{msg}, \sigma^\text{L}, \text{pk}^{\text{cmpl}}, t)$: Computes $B_{\text{msg}} = g_\lambda(\text{H}(\text{prm}_{\text{msg}}, t, \text{msg})) \subseteq [e]$ and, parse $\sigma^\text{L} = (x_k)_{k \in B_{\text{msg}}}$ and $\text{pk}^{\text{cmpl}} = (y_k)_{k \notin B_{\text{msg}}}$, and output the full candidate key $\text{pk}^{\text{L}' } = (y'_1, \dots, y'_e)$, where

$$y'_k = \begin{cases} \text{H}(\text{prm}_{\text{LOTS}}, t \| k, x_k), & k \in B_{\text{msg}}, \\ y_k, & k \notin B_{\text{msg}}. \end{cases}$$

- $0/1 \leftarrow \text{LVerify}(\text{prm}, \text{pk}^\text{L}, \text{msg}, \sigma^\text{L}, t)$: Computes B_{msg} and y'_k for $k \in B_{\text{msg}}$ as in LDerivePK . Accepts the signature σ^L iff $y'_k = \text{pk}^\text{L}[k]$ for all $k \in B_{\text{msg}}$.

Cover-freeness guarantees that for any $h' \neq h$, the block $g_\lambda(h')$ contains some index $j \notin g_\lambda(h)$, and collision-resistance of H ensures the inputs of g_λ are distinct for distinct messages. A signature on msg therefore never reveals x_j , and since H is one-way, an adversary who has only seen a signature on msg cannot recover x_j from the public value $y_j = \text{H}(\text{prm}_{\text{LOTS}}, t \| j, x_j)$. Thus, no signature on msg can be repurposed into a forgery on any other message msg' . A formal proof from the two security properties given is in the appendix, but follows [5].

Winternitz one-time signature scheme. WOTS is designed for lightweight signing and verification. WOTS^+ , one of the variants of WOTS, is used as a core component in standardized hash-based signature schemes such as XMSS and SPHINCS⁺. It is more compact than LOTS but harder to adapt to threshold cryptography; we will not reference or modify its internal functions compared to [5], so we defer a full description to Appendix A.2. We will reference it as $\text{WOTS}^+ = (\text{WKeyGen}, \text{WSign}, \text{WDerivePK}, \text{WVerify})$.

2.2 Merkle Trees

We assume a basic familiarity with Merkle trees; see Appendix A.3 for further details. We will use the following syntax:

- $\text{MT} \leftarrow \text{MerkleTree}(\text{id}_{\text{MT}}, (x_0, \dots, x_{n-1}))$: Creates a Merkle tree MT from data $x_0, \dots, x_{n-1}$. The value id_{MT} is a parameter that acts as domain separation for the internal hash function.
- $\text{R} \leftarrow \text{MT.Root}$: Outputs the root of the Merkle tree.
- $\text{path}_i \leftarrow \text{MT.Path}(i)$: Outputs an authentication path from x_i to the root of the tree (i.e., all sibling nodes of the path). In our syntax an authentication path indicates the position of leaf node i in the tree.
- $\text{R}' \leftarrow \text{MerkleTree.DeriveRoot}(\text{id}_{\text{MT}}, x, \text{path})$: Given a leaf value x and an authentication path path , derives the candidate root R' by hashing upward along the path.

We will use a salted Merkle tree $\text{MerkleTree}^{\text{Salted}}$, which salts each input x_i before hashing it into the main tree. The salt of the i th value can be accessed by $\text{MT.Salt}(i)$, and $\text{MerkleTree}^{\text{Salted}}.\text{DeriveRoot}(\text{id}_{\text{MT}}, x, \text{path}, \text{salt})$ derives the candidate root analogously, using salt for the leaf.

2.3 Batch Signatures

To aggregate one-time signatures into a multi-message signature scheme, we will effectively sign the public key of each OTS with a different long-term signature scheme (one less amenable to thresholdizing). It will be convenient to “batch” the signing. We follow [2] for the syntax and the main construction. A stateful batch signature scheme is a tuple $(\text{BKeyGen}, \text{BSign}, \text{BVerify})$ with the syntax:

- $(\text{sk}^{\text{B}}, \text{pk}^{\text{B}}) \leftarrow \text{BKeyGen}(1^\lambda)$: Generates a secret and public key pair $(\text{sk}^{\text{B}}, \text{pk}^{\text{B}})$.
- $(\text{sk}^{\text{B}'}, (\text{Sig}_i)_{i=0}^{n-1}) \leftarrow \text{BSign}(\text{sk}^{\text{B}}, (\text{msg}_i)_{i=0}^{n-1})$: Generates n signatures, corresponding to a list of n messages, and updates the secret key to $\text{sk}^{\text{B}'}$.
- $0/1 \leftarrow \text{BVerify}(\text{pk}^{\text{B}}, \text{msg}, \text{Sig})$: Verifies a single signature.

The correctness of this signature scheme is that for any $(\text{sk}^{\text{B}}, \text{pk}^{\text{B}}) \leftarrow \text{BKeyGen}(1^\lambda)$, if $\text{sk}^{\text{B}'}$ is the output of any sequence of BSign requests, it must hold that if $(\text{sk}^{\text{B}''}, (\text{Sig}_i)_{i=0}^{n-1}) \leftarrow \text{BSign}(\text{sk}^{\text{B}'}, (\text{msg}_i)_{i=0}^{n-1})$ with $\text{sk}^{\text{B}''} \neq \perp$, then $\text{BVerify}(\text{pk}^{\text{B}}, \text{msg}_i, \text{Sig}_i) = 1$ for all $0 \leq i \leq n-1$.

Any typical signature scheme security definition will suffice, e.g., eUF-CMA, by giving an adversary access to the BSign oracle.

Hypertree Batch Signatures Here we describe the batch signature of [2], which builds a Merkle tree out of the messages in each batch, then signs the root.

To authenticate the root of the Merkle tree containing the batch messages, we use a folklore construction based on nested Merkle trees, which is effectively a stateful Goldreich signature scheme. It has two parameters, L and h , and consists of h layers of Merkle trees of width L each. Each Merkle tree has WOTS^+ public keys as leaves, which are pseudorandomly generated from a long-term seed based on their position in the hypertree. To sign a batch of messages, first the messages are compressed in their own Merkle tree, then the root of this tree is signed by a leaf WOTS^+ node. The public key of this node is combined with its neighbours in a Merkle tree, and the root of this tree is signed by a leaf WOTS^+ node in the corresponding upper layer, and so forth. The final root is the long-term public key.

This can sign up to L^h batches of messages. Even with modest values of the parameters (e.g. $L = 2^{16}$, $h = 4$) it is nearly limitless for practical purposes, especially considering that in our use case the batches will themselves consist of 2^{16} LOTS public keys.

This *stateful* batch signing algorithm is nearly identical to SPHINCS⁺; the main difference is that it keeps state, and hence the signatures are smaller. We

make this tradeoff because we already assume there is a trusted key distributor, and because the threshold scheme must be stateful anyway.

2.4 Shamir Secret Sharing

Shamir's (T, N) -threshold secret sharing scheme [52] allows a dealer $\mathcal{D} \notin \mathcal{P} = \{P_1, \dots, P_N\}$ to distribute a secret $s \in \mathcal{S}$ among the participants such that any set of at least T parties can reconstruct s , whereas any set of fewer than T parties learns no information about it. The scheme provides *perfect privacy* under uniform randomness and private channels. Let $\mathcal{S} = \mathbb{Z}_p$, where $p > N$ is prime, and associate each party P_i with the nonzero point $i \in \mathbb{Z}_p$. The scheme consists of two algorithms:

- *Sharing*. On input $s \in \mathbb{Z}_p$, the dealer samples $a_1, \dots, a_{T-1} \xleftarrow{\$} \mathbb{Z}_p$ and defines $f(x) = s + a_1x + \dots + a_{T-1}x^{T-1} \in \mathbb{Z}_p[x]$. For each $i \in [N]$, it privately sends the share $f(i)$ to P_i .
- *Recombination*. For any $S \subseteq [N]$ with $|S| = T$, the secret is recovered as $s = \sum_{i \in S} f(i) \lambda_{i,S}$, where $\lambda_{i,S} = \prod_{m \in S \setminus \{i\}} \frac{m}{m-i} \in \mathbb{Z}_p$.

Algorithm 2 generalizes this to *component-wise* sharing and reconstruction of a vector $\mathbf{z} = (z_1, \dots, z_\delta) \in \mathbb{Z}_p^\delta$ of arbitrary dimension δ (taking $\delta = 1$ recovers scalar sharing above): each coordinate z_c , $c \in [\delta]$, is shared independently using a random polynomial f_c of degree $T - 1$ satisfying $f_c(0) = z_c$, and party i 's share $L_i = (f_1(i), \dots, f_\delta(i)) \in \mathbb{Z}_p^\delta$ bundles all δ evaluations at i into one vector. Rather than sampling the coefficients of each f_c directly, one may choose $T - 1$ distinct participant indices and sample the corresponding share vectors uniformly from $\mathbb{Z}_p^\delta$. Together with $L_0 = \mathbf{z}$, these T evaluations uniquely determine every f_c , yielding the same distribution as standard Shamir sharing. The remaining share vectors are then obtained by Lagrange interpolation using **ShamirReconVec**, which applies the same interpolation coefficients component-wise. More generally, the explicit argument $\{(i, L_i)\}_{i \in \mathcal{C}}$, with $\mathcal{C} \subseteq [N]$ and $|\mathcal{C}| \leq T - 1$, allows up to $T - 1$ share vectors to be fixed in advance. The algorithm then samples only $T - 1 - |\mathcal{C}|$ additional share vectors, so that a coalition $\mathcal{C}$ of corrupted parties may supply its own shares. We use this form later when key generation must accommodate adversarially chosen shares of corrupted parties.

ShamirReconVec interpolates the unique polynomial of degree at most $T - 1$ through the supplied shares and evaluates it at the point x . For vector-valued shares, the same interpolation coefficients are applied component-wise. The procedure ignores the number of shares; if they are inconsistent, it will interpolate some random value. Our protocol will notice and fix this at a higher level, so we do not enforce consistency here.

3 Threshold Signature Scheme with Identifiable Abort

At a high level, our (T, N) -threshold signature scheme, denoted TLOTS, enables any T of the N participants to jointly produce signatures under a single

Algorithm 1 Stateful batch signing with a WOTS hypertree

BKeyGen(1^λ):

- 1: $\rho \xleftarrow{\$} \{0, 1\}^\lambda$ $\triangleright$ random seed
- 2: $\text{id}_{\text{MT}} \xleftarrow{\$} \{0, 1\}^\lambda$ $\triangleright$ random parameter
- 3: **for** $i = 0$ to $L - 1$ **do**
- 4: $\text{sk}_i^W, \text{pk}_i^W \leftarrow \text{WKeyGen}(\rho, h \| i)$
- 5: **end for**
- 6: $\text{MT}_h \leftarrow \text{MerkleTree}(\text{id}_{\text{MT}} \| h \| 0, (\text{pk}_i^W)_{i=0}^{L-1})$
- 7: $\text{pk}^B \leftarrow (\text{MT}_h.\text{Root}, \text{id}_{\text{MT}})$
- 8: **return** $(\text{sk}^B = (\rho, \text{counter} = 0, \text{id}_{\text{MT}}), \text{pk}^B)$

BSign($\text{sk}^B = (\rho, \text{counter}, \text{id}_{\text{MT}}), (\text{msg}_i)_{i=0}^{n-1}$):

- 1: **if** $\text{counter} \geq L^h$ **then return** $(\text{sk}^{B'} = \perp, \perp)$ $\triangleright$ initial counter is 0
- 2: **end if**
- 3: $\text{id}_{\text{msg}} \xleftarrow{\$} \{0, 1\}^\lambda$
- 4: $\text{MT}_0 \leftarrow \text{MerkleTree}^{\text{Salted}}(\text{id}_{\text{msg}}, (\text{msg}_i)_{i=0}^{n-1})$
- 5: $m_0 \leftarrow \text{id}_{\text{msg}} \| \text{MT}_0.\text{Root}$
- 6: $\text{index} \leftarrow \text{counter}$
- 7: **for** $r = 1$ to h **do**
- 8: **for** $j = 0$ to $L - 1$ **do**
- 9: $\text{sk}_j^W, \text{pk}_j^W \leftarrow \text{WKeyGen}(\rho, r \| (j + L \lfloor \frac{\text{index}}{L} \rfloor))$
- 10: **end for**
- 11: $\text{MT}_r \leftarrow \text{MerkleTree}(\text{id}_{\text{MT}} \| r \| \lfloor \frac{\text{index}}{L} \rfloor, (\text{pk}_j^W)_{j=0}^{L-1})$
- 12: $m_r \leftarrow \text{MT}_r.\text{Root}$
- 13: $\text{path}_r \leftarrow \text{MT}_r.\text{Path}(\text{index} \bmod L)$
- 14: $t_r \leftarrow r \| \text{index}$
- 15: $\sigma_r^W \leftarrow \text{WSign}(\text{sk}_{\text{index} \bmod L}^W, m_{r-1}, t_r)$
- 16: $\text{index} \leftarrow \lfloor \frac{\text{index}}{L} \rfloor$
- 17: **end for**
- 18: $\Sigma \leftarrow (\sigma_r^W, \text{path}_r)_{r=1}^h$
- 19: **for** $i = 0$ to $n - 1$ **do**
- 20: $\text{Sig}_i \leftarrow (\text{counter}, \text{id}_{\text{msg}}, \text{MT}_0.\text{Path}(i), \text{MT}_0.\text{Salt}(i), \Sigma)$
- 21: **end for**
- 22: **return** $(\text{sk}^{B'} = (\rho, \text{counter} + 1, \text{id}_{\text{MT}}), (\text{Sig}_i)_{i=0}^{n-1})$

BVerify($\text{pk}^B = (\text{Root}, \text{id}_{\text{MT}}), \text{msg}, \text{Sig}$):

- 1: Parse $\text{Sig} = (\text{counter}, \text{id}_{\text{msg}}, \text{path}, \text{salt}, \Sigma = (\sigma_r^W, \text{path}_r)_{r=1}^h)$
- 2: **if** $\text{counter} \geq L^h$ **then return** 0
- 3: **end if**
- 4: $m_0 \leftarrow \text{id}_{\text{msg}} \| \text{MerkleTree}^{\text{Salted}}.\text{DeriveRoot}(\text{id}_{\text{msg}}, \text{msg}, \text{path}, \text{salt})$
- 5: $\text{index} \leftarrow \text{counter}$
- 6: **for** $r = 1$ to h **do**
- 7: $t_r \leftarrow r \| \text{index}$
- 8: $\text{pk}_r^W \leftarrow \text{WDerivePK}(m_{r-1}, \sigma_r^W, \text{pk}^{\text{cmpl}}, t_r)$ $\triangleright \text{pk}^{\text{cmpl}} = \emptyset$
- 9: $\text{index} \leftarrow \lfloor \frac{\text{index}}{L} \rfloor$
- 10: $m_r \leftarrow \text{MerkleTree}.\text{DeriveRoot}(\text{id}_{\text{MT}} \| r \| \text{index}, \text{pk}_r^W, \text{path}_r)$
- 11: **end for**
- 12: **return** 1 iff $m_h = \text{Root}$

Algorithm 2 Component-wise Shamir sharing and reconstruction of $\mathbf{z} \in \mathbb{Z}_p^\delta$, for arbitrary dimension δ .

ShamirShareVec($\mathbf{z}, T, N, p, \delta, \{(i, L_i)\}_{i \in \mathcal{C}}$): $\triangleright \mathcal{C} \subseteq [N], |\mathcal{C}| \leq T - 1$; default $\mathcal{C} = \emptyset$

- 1: $L_0 \leftarrow \mathbf{z}$
- 2: $S \leftarrow \mathcal{C} \cup \{0\}$
- 3: **while** $|S| < T$ **do**
- 4: Select random $i \in [N] \setminus S$
- 5: $L_i \xleftarrow{\$} \mathbb{Z}_p^\delta$
- 6: $S \leftarrow S \cup \{i\}$
- 7: **end while**
- 8: **for** all $j \in [N] \setminus S$ **do** $L_j \leftarrow \text{ShamirReconVec}(\{(i, L_i)\}_{i \in S}, S, p, j)$
- 9: **end for**
- 10: **return** $(L_1, \dots, L_N)$

ShamirReconVec($\{(i, L_i)\}_{i \in S}, S, p, x$): $\triangleright S \subseteq \{0\} \cup [N], |S| = T$; default $x = 0$;
operates component-wise if $L_i \in \mathbb{Z}_p^\delta$

- 1: **for** $i \in S$ **do** $\lambda_{i,S}(x) \leftarrow \prod_{m \in S \setminus \{i\}} \frac{m - x}{m - i} \pmod{p}$
- 2: **end for**
- 3: **return** $L \leftarrow \sum_{i \in S} \lambda_{i,S}(x) L_i$

long-term public key $\mathbf{pk}^B = (\text{Root}, \text{id}_{\text{MT}})$, using only one-way functions. The LOTS secret keys are Shamir-shared and distributed to the participants, along with the corresponding public and verification information required for signing and identifiable abort. A bulletin board, described in Appendix B, coordinates the message and one-time-key index used by the participants, thereby preventing reuse of the same one-time key for different messages. We give a high-level overview of the scheme below, focusing on a single one-time keypair; the full protocol and its algorithms are given in Section 5, where some details differ, in particular the exact contents of a signature.

- **Key generation.** The trusted dealer initializes the stateful WOTS⁺-based batch-signing hypertree described in Section 2, and announces $\mathbf{pk}^B = (\text{Root}, \text{id}_{\text{MT}})$ as the long-term public key of TLOTS. It then repeatedly generates batches of L fresh LOTS keypairs and authenticates their public keys using the batch-signing scheme.
- **Key distribution.** For each LOTS secret key $\mathbf{sk}^L = (x_1, \dots, x_e)$, the dealer Shamir-shares it among the N participants. The participant P_j receives its share vector $(\text{sh}_{j,1}, \dots, \text{sh}_{j,e})$, together with the corresponding public information and a verification-key vector $\mathbf{vk} = (\mathbf{vk}_1, \dots, \mathbf{vk}_N)$, where $\mathbf{vk}_j$ is committed to the share vector P_j , i.e. $(\text{sh}_{j,k})_{k=1}^e$. The public information includes the LOTS public key $\mathbf{pk}^L$ and its batch signature, which authenticates $\mathbf{pk}^L$ under the long-term public key $\mathbf{pk}^B$. By the privacy of Shamir secret sharing, any coalition of fewer than T participants learns no information about the secret-key components, whereas any T participants can reconstruct the

required components of $\mathbf{sk}^L$. If T partial signatures fail to reconstruct a valid signature, the verification keys are used to validate the participants' contributions and identify malformed ones, providing identifiable abort.

- **Threshold signing.** To sign a message $\mathbf{msg}$ using the designated one-time keypair, each participating party P_j computes the corresponding block $B_{\mathbf{msg}} \subseteq [e]$, with $|B_{\mathbf{msg}}| = \kappa$, and outputs the partial signature

$$\sigma_j^L = (\mathbf{sh}_{j,k})_{k \in B_{\mathbf{msg}}}.$$

Thus, each participant reveals only the κ shares indexed by $B_{\mathbf{msg}}$, while the remaining $(e - \kappa)$ shares remain hidden.

- **Combination.** An aggregator collects partial signatures from a set S of T participants and reconstructs the κ selected components of $\mathbf{sk}^L = (x_1, \dots, x_e)$ via Lagrange interpolation,

$$x_k = \sum_{j \in S} \lambda_{j,S} \mathbf{sh}_{j,k}, \quad k \in B_{\mathbf{msg}}.$$

It then outputs $\sigma^L = (x_k)_{k \in B_{\mathbf{msg}}}$, together with the corresponding public key $\mathbf{pk}^L$ and its batch signature, yielding an ordinary LOTS signature authenticated under the long-term public key. Since reconstruction is performed only for indices in $B_{\mathbf{msg}}$, the remaining $e - \kappa$ secret-key components are never reconstructed or revealed.

- **Verification.** A verifier checks the reconstructed LOTS signature under $\mathbf{pk}^L$ and verifies its batch-signature authentication path up to the long-term public key $\mathbf{pk}^B$.
- **Identifiable abort.** When a partial signature on a message $\mathbf{msg}$ fails to combine into a valid final signature on $\mathbf{msg}$, any participant P_j broadcasts the tweaked hash values of $(\mathbf{sh}_{j,k})_{k \notin B_{\mathbf{msg}}}$, letting anyone recompute the potential $\mathbf{vk}'_j$ from these hash values and the P_j 's disputed partial signature, and check if $\mathbf{vk}'_j = \mathbf{vk}_j$. An honest P_j always passes this check and a participant who submitted a malformed partial signature cannot. This gives the protocol *identifiable abort*: once an honest party collects T such proofs, it can discard the dishonest shares, name the responsible participant, and reconstruct a valid signature from the rest.

4 Security Model

We will work in the UC framework [13], though we consider a communication model more similar to the simplified universal composability framework of [14]. We describe in simple terms how this framework behaves.

We will prove security by comparison to an *ideal functionality*, where the ideal functionality is secure by definition (in our case, by returning “False” for all signatures not produced by the signing parties). We need to prove that for any adversary **Adv** against our protocol, we can write a simulator **Sim** such that for any *environment* UC.Env , the environment cannot distinguish whether it is interacting with the real adversary **Adv** and the real protocol, or our simulator **Sim** and the ideal functionality.

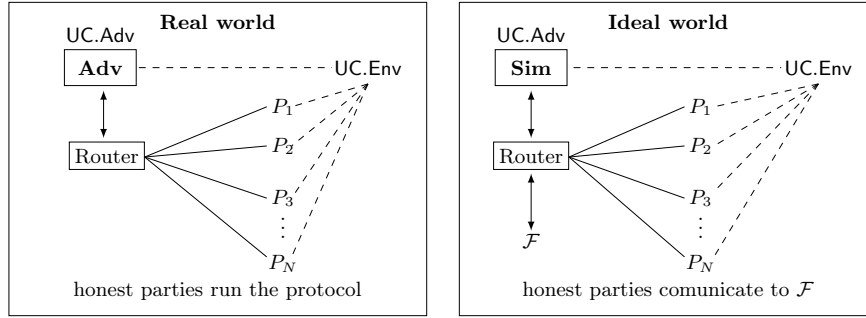


Fig. 1: Real/ideal communication model.

4.1 Communication Model

We describe a communication model which we assume the real and ideal protocol follow; we do not describe formal functionalities for this, nor explicit methods to achieve these functionalities with typical cryptographic protocols like authenticated encryption.

To avoid trivial methods of winning this game, the environment UC.Env is restricted in its interactions: it can communicate freely with the “adversary” UC.Adv (either **Adv** or **Sim**), and can also send messages to any honest party and read all outputs from honest parties. These interactions are represented by dashed lines in Figure 1.

However, all honest parties have access to a “router” channel, which is controlled by UC.Adv but invisible to UC.Env . UC.Adv has full control over the delivery of messages through the router, but cannot modify their content, and if they are marked “private” the adversary cannot read their content either. That is, the router will store the message with some unique identifier, send that ID, the recipient, sender ID, and message length to the adversary, and deliver the message once the adversary sends an instruction to deliver that message.

We assume all messages start with a function label, which we denote with bold font, e.g. **Sign** or **SignFailed**. On receiving such a message, users run the corresponding function. We will ambiguously use the same notation for the function itself and the label included in a message. This means any function which can be triggered by an external user comes with an unforgeable “senderID” indicating the sender, which we include at the end of a function’s arguments if we need it, or we suppress it for simplicity.

In the ideal functionality, honest parties do not follow the protocol but instead simply forward all messages back and forth to the ideal functionality. These messages are also invisible to the environment, making it appear to the environment as though the honest party is simply doing whatever the ideal functionality asks of them. Messages between honest parties and the ideal functionality still travel through the router, allowing them to be delayed or re-ordered by the adversary or simulator. See Figure 1.

4.2 Ideal Functionality

Our ideal functionality, Figure 2, is inspired by [15] but with some modifications. We describe our rationale below.

Initialization: Since our real protocol is parameterized by a signing threshold T and a maximum corruption threshold F , we include them separately. We must also include an initial signature limit M as input to initialization.

Key Generation: As in other UC signatures, we allow the adversary to supply the public key. The adversary also supplies a verification algorithm. Unlike [15] we separate these components, purely for ease of presentation.

Signing: The signing ideal functionality treats signing “sessions” on a first-come, first-served basis: the first user to send **sign** with any message reserves that value of **sigid** for that message. This avoids some trivial inconsistencies around **sigid**.

We deviate slightly from other threshold signing procedures because we do not need a fixed signing set: as long as at least T players approve the signature, it should be signed. Moreover, after T players approve it, others can also signal their approval to sign the same message.

Once enough players have approved the signature, the ideal functionality needs a signature. The adversary supplies one. Here, there is a tricky issue around the consistency of the scheme: our scheme allows adversarial players to block the signing process by sending invalid shares, but this is entirely detectable. Hence, the adversary is allowed to return a signature with an argument β indicating that it is an invalid signature – but if it does, it must also supply the ID of a corrupt party, who is, effectively, publicly blamed by the ideal functionality. This captures the security goal of identifiable abort.

A remaining annoyance is that thanks to out-of-order delivery, honest players may not agree on whether the shares are consistent or not. The adversary could delay and reorder messages such that one honest user receives only valid partial signatures, and another one receives invalid partial signatures. Thus, our ideal functionality accommodates multiple signatures for a given $(\text{sigid}, \text{msg})$ pair, though at most one can be valid, and the validity is fixed by the signature itself.

If the adversary attempts to supply a signature before enough players have approved it, it is rejected, unless the global state is compromised.

Consistency is enforced by the structure of the signing phase: a record with $(\text{sigid}, \text{msg}, \sigma, \beta)$ cannot be updated to a new value of β once it is recorded.

Verification: The verification checks the existing signature records. If the supplied public key matches this instance, the global state is uncompromised, but no record exists of this message, then it returns 0. This enforces the requirement that only messages receiving approval from at least T participants should be valid.

If there is a record of the message, which only occurs after at least T signing approval messages are sent to $\mathcal{F}_{\text{Sig}}$, the verification simply calls the previously-supplied $\mathcal{V}$. Since the adversary supplies this, they could effectively make all

Fig. 2: Ideal M -time Threshold Signature Functionality $\mathcal{F}_{\text{Sig}}$ **Initialization:**

On receiving **(request, sid, $F, T, M, \mathcal{P}$)** from any user P_i , if $P_i \in \mathcal{P}$ set global variable **uncompromised**, initialize all signatories in $\mathcal{P}$ as **uncompromised**, initialize **signatureMax** to M , and send a **(initialize, sid, $F, T, M, \mathcal{P}$)** message to UC.Adv. Else, ignore the message.

Key Generation:

- On receiving **(keygen, sid)** from any $P_i \in \mathcal{P}$, send **(keygen, sid, i)** to UC.Adv
- On receiving **(initialize, $\mathcal{V}, \text{pk}$)** from UC.Adv, if **pk** is not already stored, store **(sid, $\mathcal{V}, \text{pk}$)**, send **(pubkey, pk, sid)** to all $P_i \in \mathcal{P}$, and return.
- On receiving **(refresh, $\mathcal{V}, \text{pk}, M$)** from UC.Adv, if **pk** is not already stored, halt. Otherwise, set **signatureMax** $\leftarrow$ **signatureMax** + M .

Signing:

1. On receiving **(sign, sigid, msg)** from P_i :
 - If a previous **sign** message has been received with the same **sigid** but a different **msg**, end the activation
 - If no such “request” record exists, create a record **(sigid, msg, request, $\emptyset$)**
 - For the record **(sigid, msg, request, S)**:
 - Add i to S
 - If $|S| \geq T$ and the number of signature records is less than **signatureMax**, create a record **(signature, sigid, msg, $\perp, \perp, \emptyset$)** unless a record **(signature, sigid, msg, $*, *, *$)** already exists
 - Send **(sign, sigid, msg, i)** to UC.Adv.
2. On receiving **(signature, sigid, msg, σ, β, j)** from UC.Adv

If no record **(signature, sigid, msg, $*, *, *$)** exists and the global state is **uncompromised** then return. Otherwise:

 - If $\beta = 1$:
 - (a) If **(signature, sigid, msg, $\sigma, *, *$)** is already recorded or the number of recorded signatures is greater than **signatureMax**, then end the activation
 - (b) Record **(signature, sigid, msg, $\sigma, \beta, \mathcal{C} = \emptyset$)**
 - (c) Send **(signature, sigid, σ)** to all $P_i \in \mathcal{P}$
 - Else:
 - (a) If **(signature, sigid, msg, $\sigma, 1, *$)** is already recorded, end the activation
 - (b) If no record **(signature, sigid, msg, $\sigma, 0, *$)** exists, record **(signature, sigid, msg, $\sigma, 0, \mathcal{C} = \emptyset$)**
 - (c) If j is not a corrupted party, set j to the index of the first corrupted party (or $\perp$ if not found)
 - (d) Add j to the set $\mathcal{C}$ in the record for **(signature, sigid, msg, $\sigma, 0, \mathcal{C}$)**
 - (e) Send **(failure, sigid, j)** to all $P_i \in \mathcal{P}$

Verification:

Upon receiving **(verify, msg, σ, pk)** from a party q , send it to UC.Adv and do:

- Look up the record **(sid, $\mathcal{V}, \text{pk}$)** by **pk**; if no such record exists, return **(msg, $\sigma, 0$)**
- If the global state is **uncompromised**, and there is no record **(sigid, msg, $\dots$)**, return **(msg, $\sigma, 0$)**
- Else record and output **(msg, $\sigma, \mathcal{V}(\text{pk}, \text{msg}, \sigma)$)**

Corruption:

1. On receiving input **(corrupt, P_j)** from UC.Adv, record P_j as **corrupted**. If more than F signatories are corrupted, set the global state to **compromised**.
2. On receiving **(status)** from P_j , return its corruption status to P_j

signatures output “Valid”, but as this only applies to signatures which receive approval from at least T parties, this is the expected behaviour.

Corruption: The adversary can corrupt users but cannot decorrupt them. While our scheme possesses some capacity for forward secrecy (via the new key material sent during a “Key refresh”), it is limited and we choose not to formalize this.

5 Signature With Bulletin Board

Here we give our full protocol. We divide it into the functions which provide the external interface which matches the ideal functionality, internal communication functions, and internal helper functions. We also rely on two internal functionalities: (i) a key generation and (ii) a bulletin board. The bulletin board provides consensus on signing requests and we will also give an explicit construction to fit our protocol; see Appendix B. The key generation represents a key generation by a trusted third party, though by referencing an ideal functionality, we ensure that our protocol could use a distributed key generation instead.

All users running the protocol maintain an internal local state, referenced by **state**. They each know their own ID, as **id**. Occasionally we will abuse notation and add elements to uninitialized variables in **state**; assume that the user initializes the required variable to an empty list/dictionary/set and adds the required element.

We rely on a tweaked hash function, and we will deploy it with several different parameters. We will use the notation

$$H_X(t, m) = H(\text{TLOTS.pk} \parallel X, t, m) \quad (1)$$

where we use X as a string literal for domain separation and $\text{TLOTS.pk} = \text{pk}^B$, with $\text{pk}^B = (\text{MT}_h.\text{Root}, \text{id}_{\text{MT}})$, denoting the long-term public key of TLOTS.

First, we give the algorithms that are externally accessible and match the ideal functionality. These algorithms invoke several internal functions. We give a high-level explanation of each of these:

TLOTS.Initialize: As in the ideal functionality, any user can call this. It activates two internal mechanisms, the bulletin board and the key generation.

We give our ideal key generation functionality in Figure 3.

TLOTS.KeyGen: This is effectively a wrapper for $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$; the users send their approval of key generation to this ideal functionality. When it returns the key, the users will run **TLOTS.ReceiveKeys** and install the new keys.

We emphasize here a confusing layering of real and ideal functionalities:

TLOTS.KeyGen is the real protocol, but it calls the ideal key generation protocol $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$ as a subroutine. In turn, the ideal functionality $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$ is itself parameterized by real key distribution function which is specific to TLOTS, **TLOTS.KeyDist**.

For those interested in implementation and not UC security, $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$ can be realized by a trusted third party who acts as the distributor.

TLOTS.Sign: When asked to sign a message, a user checks if the message has been posted to the bulletin board. If not, it will post it and then wait to receive a follow-up signing request. In practice, it could wait for a delivery receipt from the bulletin board and then continue; our functionality models that as environment behaviour.

Once the message is posted to the bulletin board, this guarantees consensus and prevents message mismatch attacks, so the user produces a partial signature. This is just the user’s shares of specific components of a LOTS secret key. It broadcasts the partial signature and initializes an array of partial signatures, so it can eventually reconstruct a signature.

TLOTS.Verify: A TLOTS signature has 4 components: a LOTS signature on the message itself (assembled from the partial signatures), a portion of the LOTS public key for this signature, the index of the keypair as a tweak for the Lamport signature, and a signature on the LOTS public key using the Merkle hyper-tree. Here **TLOTS.Verify** verifies all components.

Ideal Functionalities In Figure 3 we give an ideal functionality for distributed key generation, parameterized by two algorithms **KeyGen** and **IssueShares**. The “ideal” aspect of this key generation is that it securely delivers the shares of a target key generation, but the keys themselves come from a real, concrete key generation algorithm.

For TLOTS, we instantiate the target key distribution using **TLOTS.KeyDist**, defined in Algorithm 4. Because of the complicated nature of this key generation, we assume it is performed by a trusted third party at setup time; however, thanks to the composable nature of the UC framework, securely realizing this functionality with a distributed protocol would be securely compatible with TLOTS.

We will also use this ideal key generation as part of the bulletin board, albeit with a different target key generation algorithm.

Internal Communication Here we describe, at a high level, the functions run for internal communication among the users during signing. Algorithm 5 describes the algorithms precisely.

TLOTS.ReceiveKeys: This is run in response to being delivered keys by $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$. Mainly it installs keys; however, it must also check if there were messages to be signed that have not yet been signed because there were not enough keys. If so, it proceeds through the same logic as **TLOTS.Sign**: it checks that it has enough keys to sign, then creates and broadcasts a partial signature.

TLOTS.Sign₂: This function is run when a user receives a broadcasted partial signature. Firstly, if the user has seen a valid signature for this sigid before, it ignores the message. Second, it stores the partial signature. Once it has enough partial signatures, it attempts to recover a full signature.

Algorithm 3 Externally accessible TLOTS algorithms matching $\mathcal{F}_{\text{Sig}}$. These are run by a participant with local variables in **state** and ID **id**.

TLOTS.Initialize($\text{sid}, F, T, M, \mathcal{P}$)

- 1: Send (**BB.Activate**, $F, T, \mathcal{P}$) to $\mathcal{F}_{\text{BB}}$
- 2: Send ($\text{sid}, F, T, M, \mathcal{P}$) to $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$
- 3: Store ($\text{sid}, F, T, M, \mathcal{P}$) in **state**

TLOTS.KeyGen($\text{sid}, F, T, \mathcal{P}$)

- 1: If ($\text{sid}, F, T, \mathcal{P}$) not already stored, store them in **state**
- 2: Send (**KeyGen**, sid) to $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$

TLOTS.Sign(sigid, msg)

- 1: $\text{index} \leftarrow \text{BB.Search}(\text{sigid}, *)$
- 2: **if** $\text{index} \neq \perp$ **then**
- 3: **if** $\text{BB.Read}(\text{index}) \neq (\text{sigid}, \text{msg})$ **then** Return
- 4: **end if**
- 5: $\text{state.signings}[\text{sigid}] \leftarrow (\text{msg}, \text{index})$
- 6: **if** $\text{state.sk}^{\text{L}}[\text{index}] \neq \perp$ **then** $\triangleright \text{state.sk}^{\text{L}}[\text{index}] = \text{sh}_{\text{index}, \text{id}}^{\text{L}}$, where **id** is the identity of the party running the algorithm
- 7: $\sigma_{\text{sigid}, \text{id}}^{\text{L}} \leftarrow \text{TLOTS.PartialSignature}(\text{state.sk}^{\text{L}}[\text{index}], \text{msg}, \text{sigid})$
- 8: Broadcast (**TLOTS.Sign₂**, $\text{sigid}, \sigma_{\text{sigid}, \text{id}}^{\text{L}}$) to all $P_i \in \mathcal{P} \setminus P_{\text{id}}$
- 9: $\text{state.partialSignatures}[\text{sigid}][\text{id}] \leftarrow \sigma_{\text{sigid}, \text{id}}^{\text{L}}$ $\triangleright$ **id** is the ID of the user running this function
- 10: **else** $\triangleright$ not enough key material
- 11: Send (**KeyGen**, sid) to $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$
- 12: **end if**
- 13: **else**
- 14: $\text{state.signings}[\text{sigid}] \leftarrow (\text{msg}, \perp)$
- 15: **BB.Post**($\text{sigid}, \text{msg}, \mathcal{P}$)
- 16: **end if**

TLOTS.Verify($\text{pk}^{\text{B}}, \text{msg}, \sigma$) $\triangleright \text{pk}^{\text{B}} = (\text{MT}_h.\text{Root}, \text{id}_{\text{MT}})$ is the long-term public key of TLOTS

- 1: Parse σ as $(\sigma^{\text{L}}, \text{pk}^{\text{cmpl}}, \text{index}, \text{Sig})$ $\triangleright$
 $\text{Sig} = (\text{counter}, \text{id}_{\text{msg}}, \text{MT}_0.\text{Path}(i), \text{MT}_0.\text{Salt}(i), \Sigma)$
- 2: $\text{pk}^{\text{L}} \leftarrow \text{LDerivePK}(\text{prm}, \text{msg}, \sigma^{\text{L}}, \text{pk}^{\text{cmpl}}, \text{index})$
- 3: **if** $\text{BVerify}(\text{pk}^{\text{B}}, \text{pk}^{\text{L}}, \text{Sig}) = 1$ **then** return 1; $\triangleright \text{pk}^{\text{L}}$ is as **msg** in Algorithm 1
- 4: **else** return 0;
- 5: **end if**

Fig. 3: Ideal M -use Distributed Key Generation $\mathcal{F}_{\text{DKG}}^{\text{KeyDist}}$ (parameterized by $\text{KeyDist} = (\text{KeyGen}, \text{KeyRefresh}, \text{IssueShares})$).

Initialization:

At first activation by any party P_i , interpret the input as $(\text{sid}, F, T, M, \mathcal{P})$, where sid is the session ID, $\mathcal{P} = \{P_1, \dots, P_N\}$ is a set of signatories, T is a threshold, F is a maximum number of corruptions, and M is the initial maximum number of signatures. Record all signatories as non-corrupted, initialize a counter $\text{ctr} = 0$, and a global state variable to **uncompromised**. Generate and store $(\text{sk}, \text{pk}) \leftarrow \text{KeyDist.KeyGen}(F, T, \mathcal{P})$.

Key-Generation

1. Upon receiving **(keygen, sid)** from a party $P_i \in \mathcal{P}$, send **(keygen, i , sid)** to UC.Adv.
2. Upon receiving $(\mathcal{C}, \{\text{sk}_i\}_{i \in \mathcal{C}})$ from UC.Adv, if all $P_i \in \mathcal{P}$ have input **keygen**, all $i \in \mathcal{C}$ have status **corrupted**
 - (a) If no tuple $(\text{pk}, \text{sk}, \text{sid})$ has been recorded, record $(\text{pk}, \text{sk}, \text{sid})$
 - (b) Else, $\text{sk} \leftarrow \text{KeyDist.KeyRefresh}(\text{sk}, M)$
 - (c) $\{\text{sk}_i\}_{i \in [N]} \leftarrow \text{KeyDist.IssueShares}(\text{sk}, T, N, p, \{(i, \text{sk}_i)\}_{i \in \mathcal{C}})$
 - (d) If $|\mathcal{C}| > F$, set $\text{sk}_i \leftarrow \perp$ for all $P_i \in \mathcal{P}$.
 - (e) Send **(ReceiveKeys, sid, pk, sk_i)** to all $P_i \in \mathcal{P}$.

Corruption:

On receiving **(corrupt, P_j)** from UC.Adv, record P_j as corrupted.

Algorithm 4 Key distribution functions for use in $\mathcal{F}_{\text{DKG}}^{\text{KeyDist}}$ (Figure 3).

TLOTS.KeyDist.KeyGen($F, T, \mathcal{P}$):

- 1: $(\text{sk}^B, \text{pk}^B) \leftarrow \text{BKeyGen}(1^\lambda)$
 - 2: **return** $(\text{sk} = (\text{sk}^B, \text{lastShared} = 0, \emptyset), \text{pk}^B)$
-

TLOTS.KeyDist.KeyRefresh($\text{sk} = (\text{sk}^B, \text{lastShared}, (\text{sk}_i^L, \text{Sig}_i)_{i=1}^n), M$):

- 1: **for** $i = n + 1$ to $n + M$ **do**
 - 2: $(\text{sk}_i^L, \text{pk}_i^L) \leftarrow \text{LKeyGen}(\text{prm}, 1^\lambda, i)$ $\triangleright \text{prm} = (\text{pk}^B \parallel \text{"LOTS"}, \text{pk}^B \parallel \text{"msg"})$
 - 3: **end for**
 - 4: $(\text{Sig}_i)_{i=n+1}^{n+M} \leftarrow \text{BSign}(\text{sk}^B, (\text{pk}_i^L)_{i=n+1}^{n+M})$
 - 5: $n \leftarrow n + M$
 - 6: $\text{lastShared} \leftarrow n - M$
 - 7: **return** $\text{sk} = (\text{sk}^B, \text{lastShared}, (\text{sk}_i^L, \text{Sig}_i)_{i=1}^n)$
-

TLOTS.KeyDist.IssueShares($F, T, \text{pk}, \text{sk} = (\text{sk}^B, \text{lastShared}, (\text{sk}_i^L, \text{Sig}_i)_{i=1}^n)$):

- 1: **for** $i = \text{lastShared} + 1$ to n **do**
 - 2: $(\text{sh}_{i,j}^L)_{j=1}^N \leftarrow \text{ShamirShareVec}(\text{sk}_i^L, T, N, p, \delta, \{(j, L_j)\}_{j \in \mathcal{C}})$ $\triangleright \text{sh}_{i,j}^L = (\text{sh}_{i,j,k}^L)_{k=1}^e$
 - 3: $((\pi_{i,j,k})_{j=1}^N)_{k=1}^e \leftarrow \text{H}_{\text{vk}}(i \parallel j \parallel k, \text{sh}_{i,j,k}^L)$ $\triangleright \text{For all } j \in [N], k \in [e]$
 - 4: $\text{vk}_i = (\text{vk}_{i,j})_{j=1}^N \leftarrow \text{H}_{\text{vk}}(i \parallel j, (\pi_{i,j,k})_{k=1}^e)_{j=1}^N$ $\triangleright \text{For all } j \in [N]$
 - 5: **end for**
 - 6: $\text{sh}_j^L \leftarrow (\text{sh}_{i,j}^L, \text{vk}_i, \text{Sig}_i)_{i=\text{lastShared}+1}^n$
 - 7: **for** $j \in [N]$ **do return** sh_j^L to P_j
 - 8: **end for**
-

This is a complicated procedure so we separated it into another function, **TLOTS.ReconstructSignature**, but essentially it broadcasts a signature if there is a success, or starts an identifiable abort phase otherwise.

TLOTS.SignFailed: When a signature fails, users broadcast proofs to each other. This function deals with these proofs. First it looks up the partial signature that was originally sent from the sender of the proof, then applies the verification process described in the introduction. If the proof fails, they broadcast a failure message identifying the misbehaving user, and record the signer as corrupt. If the proof passes, they store this fact. If they receive at least T valid proofs, then they know they can reconstruct the signature from those (effectively ignoring malformed partial signatures), so they do this and broadcast the result.

TLOTS.Signature: This function allows valid signatures to be “contagious”. When a user sees a signature, it checks if it was valid. If so, it installs the signature as a “success”, which allows it to ignore any future communication about this signature. It also outputs the signature, which is necessary to match the ideal functionality; tracking successes and failures ensures this does not create a loop of honest users broadcasting the same signature back and forth.

Internal Helper Functions These are subroutines of the previous functions, described at a high level here and detailed in Algorithm 6.

TLOTS.ReconstructSignature: This is called on sets of partial signatures. The first thing it does is attempt to reconstruct the one-time signature. Then it extracts the other associated public key data from this user’s own state and attempts to verify. If verification succeeds, it marks this **sigid** as successful and broadcasts the signature. If verification fails, it marks this **sigid** as a failure and starts the identifiable abort. To do this, it computes a proof π for its signature, then broadcasts this proof as a **TLOTS.SignFailed** message.

TLOTS.PartialSignature: This produces a partial signature; it is effectively identical to signing algorithm, but it takes *shares* as input. Notice that the input is a specific $\text{sh}_{i,\text{id}}^L$, which is only one party’s share of *one* OTS secret key (selected for a specific index).

TLOTS.SigProof: This produces a proof that a partial signature is consistent with the shares distributed to a participant. For a LOTS key at index *index*, participant P_j holds the share vector $(\text{sh}_{j,k})_{k \in [e]}$, and its verification value vk_j commits to the hashes of all these shares. A partial signature reveals $\text{sh}_{j,k}$ for all $k \in B_{\text{msg}}$; therefore, to complete the opening of vk_j , participant P_j sends $\pi = (\text{H}_{\text{vk}}(\text{index} \| j \| k, \text{sh}_{j,k}))_{k \notin B_{\text{msg}}}$ as its proof. The verifier hashes the revealed components of the partial signature, combines them with π , and recomputes vk_j . Hence, a valid proof shows that the partial signature is consistent with the shares committed to during key distribution.

Algorithm 5 Internal communication functions for TLOTS, accessible only to parties in $\mathcal{P}$ (assume each begins with a line that halts if the sender is not in $\mathcal{P}$).

TLOTS.ReceiveKeys($\text{pk}^B, \text{sh}^L; \text{senderID}$)

- 1: Halt if senderID is not $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$
- 2: Parse sh^L as an array sk^L of M shares $(\text{sh}_{i,\text{id}}^L)_{i=1}^M$ of LOTS secret keys, an array pk^L of M LOTS public keys and batch signatures, and an array vk of M verification keys
- 3: Append $\text{pk}^B, \text{sk}^L, \text{pk}^L$, and vk to the respective values in state
- 4: **if** this is the first set of keys received **then** Output ($\text{pubkey}, \text{pk}^B$)
- 5: **end if**
- 6: Let index_0 be the number of keys before, and index_1 the number of keys currently
- 7: **for** all sigid with $\text{state.signings}[\text{sigid}] = (\text{msg}, \text{index})$ such that $\text{index} > \text{index}_0$ **do**
- 8: **if** $\text{state.sk}^L[\text{index}] \neq \perp$ **then** $\triangleright \text{state.sk}^L[\text{index}] = \text{sh}_{\text{index}, \text{id}}^L$
- 9: $\sigma_{\text{sigid}, \text{id}}^L \leftarrow \text{TLOTS.PartialSignature}(\text{state.sk}^L[\text{index}], \text{msg}, \text{sigid})$
- 10: Broadcast ($\text{TLOTS.Sign}_2, \text{sigid}, \sigma_{\text{sigid}, \text{id}}^L$) to all $P_i \in \mathcal{P} \setminus P_{\text{id}}$
- 11: $\text{state.partialSignatures}[\text{sigid}][\text{id}] \leftarrow \sigma_{\text{sigid}, \text{id}}^L$ $\triangleright \text{id}$ is the ID of the user running it
- 12: **else** $\triangleright \text{still not enough key material}$
- 13: Send (KeyRefresh, id) to $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$
- 14: **end if**
- 15: **end for**

TLOTS.Sign₂($\sigma_{\text{sigid}, \text{senderID}}^L, \text{sigid}, \text{senderID}$)

- 1: **if** state.successes contains sigid **then** Return
- 2: **end if**
- 3: $(\text{msg}, \text{index}) \leftarrow \text{state.signings}[\text{sigid}]$
- 4: **if** $\text{state.partialSignatures}[\text{sigid}]$ contains senderID **then** Return $\triangleright$ Prevents multiple signatures
- 5: **end if**
- 6: $\text{state.partialSignatures}[\text{sigid}][\text{senderID}] \leftarrow \sigma_{\text{sigid}, \text{senderID}}^L$
- 7: **if** $|\text{state.partialSignatures}[\text{sigid}]| \geq T$ **then**
- 8: **TLOTS.ReconstructSignature**($\text{sigid}, \text{msg}, \text{index}, \text{partialSignatureArray}$)
- 9: **end if**

TLOTS.SignFailed($\text{sigid}, \pi, \text{senderID}$)

- 1: $(\text{msg}, \text{index}) \leftarrow \text{state.signings}[\text{sigid}]$
- 2: **if** no such element exists **then** $\text{state.proofs}[\text{sigid}].\text{add}((\text{sigid}, \pi; \text{senderID}), \text{unvalidated}) \wedge$ Return
- 3: **end if**
- 4: $\sigma_{\text{sigid}, \text{senderID}}^L \leftarrow \text{state.partialSignatures}[\text{sigid}][\text{senderID}]$
- 5: $B_{\text{msg}} \leftarrow g_{\lambda}(\text{H}_{\text{msg}}(\text{index}, \text{msg})) \subseteq [e]; \pi' \leftarrow (\perp)^e$
- 6: **for** $k \in B_{\text{msg}}$ **do** $\pi'[k] \leftarrow \text{H}_{\text{vk}}(\text{index} \parallel \text{senderID} \parallel k, \sigma_{\text{sigid}, \text{senderID}}^L[k])$
- 7: **for** $k \notin B_{\text{msg}}$ **do** $\pi'[k] \leftarrow \pi[k]$
- 8: $v \leftarrow \text{H}_{\text{vk}}(\text{index} \parallel \text{senderID}, \pi')$
- 9: **if** $\text{state.vk}[\text{index}][\text{senderID}] \neq v$ **then**
- 10: Add senderID to $\text{state.corrupt} \wedge$ Broadcast (failure, $\text{sigid}, \text{senderID}$)
- 11: **else**
- 12: Add $((\text{sigid}, \pi, \text{senderID}), \text{validated})$ to $\text{state.proofs}[\text{sigid}]$
- 13: **if** $\text{state.proofs}[\text{sigid}]$ contains at least T validated proofs **then**
- 14: $\text{partialSignatureArray} \leftarrow \text{state.partialSignatures}[\text{sigid}][i]$ for all i such that $\text{state.proofs}[\text{sigid}][i]$ is validated
- 15: **TLOTS.ReconstructSignature**($\text{sigid}, \text{msg}, \text{index}, \text{partialSignatureArray}$)
- 16: **end if**
- 17: **end if**

TLOTS.Signature(sigid, σ):

- 1: $(\text{msg}, \text{index}) \leftarrow \text{state.signings}[\text{sigid}]$
- 2: **if** state.successes contains $\text{sigid} \vee \text{TLOTS.Verify}(\text{pk}^B, \text{msg}, \sigma)$ returns False **then** Return
- 3: **end if**
- 4: **if** state.failures contains sigid **then** Remove sigid from state.failures
- 5: **end if**
- 6: Add sigid to $\text{state.successes} \wedge$ Output ($\text{signature}, \text{sigid}, \sigma$)

We remark briefly on the message flow of failed signatures. Honest users do not need to announce that a signature failed; they simply start broadcasting a proof that their share is correct. This means adversarial players could send inconsistent messages, such that one user obtains a valid signature and other users do not. This is fine: if any user obtains a set of valid shares, they create a signature and broadcast it, and users who previously constructed a failed signature will accept the valid signature and move on. If all honest users receive an invalid partial signature, they will all output proofs, and on receiving each others' (valid) proofs, they will be able to determine a set of valid partial signatures and continue.

One might ask: why not send the proofs in the first round? We propose this method as it means under normal operation, there is less communication. Since misbehaving parties can be immediately detected and caught (only escaping detection if they go offline, which is presumably dealt with at a policy level), we claim this should create enough disincentive that in normal operation we will see ongoing communication savings.

5.1 Security

Theorem 1. *TLOTS UC-securely realizes the ideal functionality $\mathcal{F}_{\text{Sig}}$. That is, for any adversary $\mathbf{Adv}$ against TLOTS there exists a simulator $\mathbf{Sim}$ such that for any environment UC.Env , the advantage of UC.Env in distinguishing the real and ideal executions is bounded by $9\text{Adv}^{\text{SM-TCR}}(\mathbf{H}) + 2\text{Adv}^{\text{SM-DSPR}}(\mathbf{H}) + \text{Adv}^{\text{eUF-CMA}}(\mathbf{B})$.*

The proof is in Section C.

6 Bulletin Board

We detail our bulletin board in Appendix B, but it is a simple idea so we sketch it here. It is a permissioned blockchain with N hosts, where each “host” maintains a list of blocks containing one message, a hash of the previous block, and a collection of signatures on the message. See Figure 5. We rely on a single “aggregator” party to relay any message that should be posted to the bulletin board to all other parties. On receiving a message from the aggregator, a host simply signs it with their long term secret key (shown by Σ^* , indexed by the block number and signer id) and returns it to the aggregator. An honest host will never sign two different messages for the same position in the blockchain. Once receiving enough signatures, the aggregator will send this finalized block back to all the hosts, who validate the signatures and add it to their blockchains. For our scheme to tolerate up to F corruptions, a host will only consider a block valid if it has at least $\frac{N+F+1}{2}$ valid signatures from other hosts. Since at most F of these signatures come from malicious hosts, at least $\frac{N-F+1}{2}$ signatures come from honest hosts. Since there are at least $N - F$ honest hosts, for any two valid blocks, there is at least one honest host who signed both. Since an honest host

Algorithm 6 Internal helper functions for TLOTS. These are not externally triggered, but called by other algorithms.

```

TLOTS.ReconstructSignature(sigid, msg, index, partialSignatureArray)
1: if |partialSignatureArray|  $\geq T$  then
2:    $S \leftarrow$  any  $T$ -subset of the party indices in partialSignatureArray
3:    $\sigma^L \leftarrow \text{ShamirReconVec}(\{(i^*, \text{partialSignatureArray}[i^*])\}_{i^* \in S}, S, p, 0)$ 
4:    $(\text{pk}^L, \text{Sig}) \leftarrow \text{state.pk}[\text{index}]$ 
5:   if LVerify(prm,  $\text{pk}^L$ , msg,  $\sigma^L$ , index) then
6:      $B_{\text{msg}} \leftarrow g_\lambda(\text{H}_{\text{msg}}(\text{index}, \text{msg}))$ 
7:      $\text{pk}^{\text{cmpl}} \leftarrow (\text{pk}_k^L)_{k \notin B_{\text{msg}}}$ 
8:      $\sigma \leftarrow (\sigma^L, \text{pk}^{\text{cmpl}}, \text{index}, \text{Sig})$ 
9:     Broadcast (signature, sigid,  $\sigma$ )
10:    state.successes.add(sigid)
11:   else
12:      $\pi \leftarrow \text{TLOTS.SigProof}(\text{state.sk}^L[\text{index}], \text{msg}, \text{index})$ 
13:     Broadcast (TLOTS.SignFailed, sigid,  $\pi$ )
14:     state.failures.add(sigid)
15:     Add ((sigid,  $\pi$ , id), validated) to state.proofs[sigid]
16:     for each unvalidated (sigid,  $\pi'$ ; senderID) in state.proofs[sigid] do
17:       TLOTS.SignFailed(sigid,  $\pi'$ ; senderID)
18:     end for
19:   end if
20: end if

TLOTS.PartialSignature( $\text{sh}_{\text{index}, \text{id}}^L$ , msg, index):  $\triangleright \text{sh}_{\text{index}, \text{id}}^L = (\text{sh}_{\text{index}, \text{id}, k})_{k=1}^e$ 
1:  $B_{\text{msg}} \leftarrow g_\lambda(\text{H}_{\text{msg}}(\text{index}, \text{msg}))$ 
2:  $\sigma_{\text{index}, \text{id}}^L \leftarrow (\text{sh}_{\text{index}, \text{id}, k})_{k \in B_{\text{msg}}}$ 
3: return  $\sigma_{\text{index}, \text{id}}^L$ 

TLOTS.SigProof( $\text{sh}_{\text{index}, \text{id}}^L$ , msg, index):
1:  $\pi \leftarrow (\perp)^e$ 
2:  $B_{\text{msg}} \leftarrow g_\lambda(\text{H}_{\text{msg}}(\text{index}, \text{msg}))$ 
3: Set  $\pi[k] \leftarrow \text{H}_{\text{vk}}(\text{index} \parallel \text{id} \parallel k, \text{sh}_{\text{index}, \text{id}, k}^L)$  for all  $k \notin B_{\text{msg}}$ 
4: return  $\pi$ 

```

never signs two messages for the same block, we achieve consistency among valid blocks.

Compared to other consensus mechanisms, this is lightweight and simple, mainly because of our trusted aggregator assumption. A malicious aggregator cannot create inconsistent valid blocks (which would break unforgeability of our one-time signatures), but a malicious aggregator can easily disrupt progress: if the aggregator sends different messages to different honest hosts this will irreparably corrupt the state of the bulletin board; if the aggregator ignores signatures or goes offline, the bulletin board shuts down. Dynamic selection of an aggregator would be a great feature for availability, but such features become complicated rather quickly. Luckily, the modularity of the UC formalism means that any implementation requiring a more reliable bulletin board can use TLOTS with any secure bulletin board it needs.

6.1 Ideal Functionality

We give a specific bulletin board functionality. Our goal is to provide eventual consensus on ordering, with no guarantees on progress. Thus we have the ideal functionality allow the adversary to receive all messages and return them in any order it pleases. Once the adversary has “finalized” a message, it remains in a fixed place in the bulletin board. Any user can read messages once they are finalized.

Our ideal functionality, Figure 4, also includes a convenience function **Search**, which returns the index of a message that has been finalized. This can be simulated in linear-time with **Read** by checking all indices.

7 Robustness

Robustness is a stronger property of a threshold signature which states that even if an adversary misbehaves arbitrarily – sending garbage, going offline, etc. – the protocol can still complete the signing process. In our communication model, this is obviously false, as the adversary has total control over message delivery and can simply halt all messages.

We do not want to get into the details of formalizing message timing and timeout to rigorously model robustness (e.g., this would require the ideal functionality to detect if an adversary fails to return a signature within a specified time). However, we make the following informal robustness claims about our protocol. By our UC security claims we know that adversarial players cannot learn the secret(s), so for robustness we only need certain parties to follow the protocol as specified. This means that, even though our bulletin board tolerates very few fully malicious players, unforgeability can tolerate many more, leaving us able to handle a large portion of players who follow the protocol as written, even if they covertly try to recover the secrets (i.e., “semi-honest”).

Theorem 2 (Informal). *TLOTS with threshold T is robust when using a robust bulletin board, as long as at least T users are honest*

FIGURE 4 Ideal bulletin board functionality $\mathcal{F}_{\text{BB}}$ **Activate:**

- On activating with a list of parties $\mathcal{P}$ and a maximum corruption limit of F , set global state to **uncompromised**. If $F > |\mathcal{P}|/3$, halt.
- Initialize an empty list B for the bulletin board, an empty list Q for a queue of messages

Post:

- On receiving a message **msg** from party P_i , add (msg, i) to Q and send (msg, i) to UC.Adv.

Finalize:

- On receiving (msg, i) from UC.Adv, if (msg, i) is in Q , remove (msg, i) from Q and add (msg, i) to the end of B ; otherwise do nothing

Read:

- If global state is **uncompromised**, on receiving i from any party, return **(response, $B[i]$)**
- Otherwise send the request and sender ID to UC.Adv. and return any reply given

Search:

- If global state is **uncompromised**, on receiving **msg** from any party, return the first index i such that $B[i] = (\text{msg}, *)$ for any user $*$.
- Otherwise send the request and sender ID to UC.Adv. and return any reply given

Corruption:

1. On receiving input **(corrupt, P_j)** from UC.Adv, record P_j as **corrupted**. If more than F signatories are corrupted, set the global state to **compromised**.
2. On receiving input **(status)** from P_j , return its corruption status to P_j .

Fig. 4: Ideal bulletin board functionality

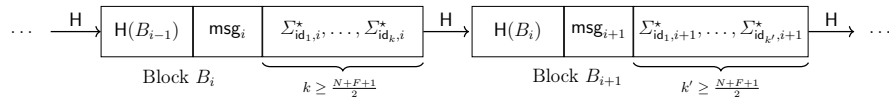


Fig. 5: Each block contains collected signatures satisfying the majority threshold, where the identities of the signers need not be the same across different blocks.

Proof. We see that if at least T honest users are requested to sign, they will post to the bulletin board. If the bulletin board is robust, this message is eventually delivered. Once the message is posted, all honest users will produce a partial signature. They will all receive each other's partial signatures. If the adversary does not contribute any malformed shares, they will immediately output a valid signature. If the adversary contributes malformed shares, they must deliver malformed shares to all honest users: if any honest user is not given a malformed

share, they will still receive valid shares from other honest users and be able to output a valid signature.

If all honest users receive malformed shares, they will all produce valid proofs, which they broadcast to each other. As soon as an honest user receives $T - 1$ valid proofs from other honest users, they can construct a set of valid partial signatures and reconstruct a valid signature.

Thus, as long as messages between honest users are eventually delivered, a signature is eventually produced. $\square$

Theorem 3 (Informal). *Our bulletin board is robust as long as at least $\frac{N+F+1}{2}$ of the users are honest, including the aggregator.*

Proof. When a bulletin board host receives a “Post” request, they forward it to the aggregator. The aggregator maintains an internally consistent ordering, which is eventually reflected in the blockchains of the other hosts. As soon as the aggregator receives a message to post, they form a new block and send it to other hosts (or add it to their signing queue; this distinction does not matter as it will eventually make it to the front of the queue). Since an honest aggregator only sends valid blocks, other honest users will sign it as soon as they receive it, and return a valid signature. The honest aggregator accepts the new block as soon as it has $\frac{N+F+1}{2}$ signatures, which it will obtain because there are at least that many honest users who will return signatures. The aggregator then sends this valid block to the other users, who validate and install it. Then they proceed to the next requested block, if it exists. $\square$

Despite the informality of these “proofs”, we feel confident in the claim of robustness. The exact details of timing will depend on the specific network assumptions we make, but there is no protocol-level way for corrupt players to arrest progress (except corrupting the bulletin board aggregator).

We also emphasize that a semi-honest aggregator (i.e., who delivers messages reliably) is a requirement for the robustness of ROAST [51].

8 Performance

We implemented TLOTS in Rust; the bulletin board of Section 6 is not benchmarked. The code is available at the commands that regenerate every figure below, is available at **Threshold Signatures with IA from OWF**.

Parameters. Hash outputs are 256 bits ($\lambda = 256$), and the tweakable hash is instantiated with SHA-256, its public parameter and tweak length-prefixed into the input so that (prm, t, x) is uniquely decodable. We take $(e, \kappa) = (261, 123)$: the smallest e with $\binom{e}{\lfloor e/2 \rfloor} \geq 2^{256}$, then the smallest κ with $\binom{e}{\kappa} \geq 2^{256}$, which also makes g_λ injective. Each component in a LOTS secret key $\text{sk}^L = (x_1, \dots, x_e)$ is split into $r = 8$ blocks of $\text{bpb} = 32$ bits, Shamir-shared independently over $\mathbb{F}_p$ with the FFT-friendly prime $p = 4101 \cdot 2^{20} + 1$, while $2^{20} \mid p - 1$ lets KeyDist evaluate at the points ω^j of a primitive $2^{\lceil \log N \rceil}$ -th root of unity, for any $N \leq 2^{20}$.

The multitree has height h and d layers of subtrees of height h/d ($d \mid h$), so $L = 2^{h/d}$ leaves per subtree: the bottom layer’s are the Shamir-shared LOTS keys, the $d-1$ above them WOTS⁺ trees. Each subtree is a deterministic function of the master seed and its (layer, subtree) address, and each leaf of its position within that subtree. We report $h = 64$, $d = 4$ ($L = 2^{16}$) for 2^{64} signatures, with $(\ell_1, \ell_2, \ell) = (64, 3, 67)$ at Winternitz parameter $w = 4$ (base $2^w = 16$, denoted $w = 16$ in XMSS).

Experimental Results. We ran all experiments on a MacBook Pro Mac14,7 (Apple M2, 4 performance + 4 efficiency cores, 16 GiB unified memory), compiled with Rust 1.89.0 in `-release` mode; the implementation is single-threaded throughout, so one core is used regardless of the machine. Table 1 reports the (T, N) -dependent costs for one leaf; every signature verified, and run-to-run variation is a few percent (three runs gave 337.0, 350.1 and 351.5 ms for `IssueShares` at $N = 1000$). `TLOTS.Verify` stays below 1 ms and `TLOTS.PartialSignature`, summed over all T signers, below 2.1 ms, both regardless of N ; a single participant’s contribution is therefore about $10 \mu\text{s}$ at $T = 200$. The NTT-based share evaluation makes `IssueShares` independent of T ($N = 100$ costs 39–44 ms at either $T = 25$ or $T = 50$, a spread within run-to-run noise). Since the Lagrange coefficients depend only on the signer set S , `TLOTS.ReconstructSignature` computes them once and reuses them across all $\kappa \cdot r$ coordinates, for a total cost of $\Theta(T^2 + \kappa \cdot r \cdot T)$. In our fault-injection test, a malformed partial signature breaks reconstruction, but at $N = 50, T = 25$ all 50 openings take 4.6 ms and checking them 5.6 ms, after which exactly the culprit is named and the round and negligible computation to the failure path, nothing to the common one.

Table 1: Threshold signing benchmarks for one leaf; times in ms.

N	T	<code>IssueShares</code>	computing vk	<code>PartialSignature</code>	Reconstruction	Verify	Total
10	5	3.357	2.016	0.301	0.098	0.886	4.641
50	25	24.575	51.864	0.275	0.144	0.806	25.798
100	25	43.956	18.991	0.280	0.218	0.749	45.204
100	50	39.158	18.952	0.583	0.233	0.748	40.721
400	100	150.998	73.311	1.002	0.530	0.808	153.337
1000	200	351.488	187.669	2.090	1.105	0.748	355.432

Sizes. The public key is $\text{pk}^B = (\text{MT}_h.\text{Root}, \text{id}_{\text{MT}})$, and a signature is $\sigma = (\sigma^L, \text{pk}^{\text{cmpl}}, \text{index}, \text{Sig})$ with $\text{Sig} = (\text{counter}, \text{id}_{\text{msg}}, \text{MT}_0.\text{Path}, \text{MT}_0.\text{Salt}, \Sigma)$. It need not carry the leaf public key pk^L , which verification recovers as $\text{LDerivePK}(\text{prm}, \text{msg}, \sigma^L, \text{pk}^{\text{cmpl}}, \text{index})$; only the $e - \kappa$ off-block hashes travel, saving $\kappa \cdot 32 = 3936$ B. Identifiable abort costs no signature bytes and the opening π travels only when a round fails. Table 2 breaks down $|\sigma|$.

Table 2: Size of pk^B and σ .

Component	Formula	Size (B)
$\text{pk}^B = (\text{MT}_h.\text{Root}, \text{id}_{\text{MT}})$	$2 \cdot 32$	64
$\sigma^L, \text{pk}^{\text{cpl}}$	$e \cdot 32$	8 352
$\text{id}_{\text{msg}}, \text{MT}_0.\text{Salt}, \text{MT}_0.\text{Path}, \text{index}$	$(h/d + 2) \cdot 32 + h/8$	584
$\Sigma: d - 1 \text{ pairs } (\sigma_r^W, \text{Path}_r)$	$(d - 1)(\ell + h/d) \cdot 32$	7 968
Total σ		16 904

Thresholds. With an external consensus protocol, any signing threshold T_{TLOTS} can be chosen, and the scheme remains UC-secure against up to $F = T_{\text{TLOTS}} - 1$ fully malicious parties. Our specific bulletin board, however, has its own threshold, so the scheme has two independent thresholds: T_{BB} for the bulletin board and $T_{\text{TLOTS}} > F$ for signing. Each block on the bulletin board receives T_{BB} signatures, and any two valid blocks must share at least one honest signer. Since two such signer sets overlap in at least $2T_{\text{BB}} - N$ parties, consistency requires $2T_{\text{BB}} - N \geq F + 1$, i.e., $T_{\text{BB}} \geq \lceil \frac{N+F+1}{2} \rceil$.

Robustness additionally requires $T_{\text{BB}} \leq N - F$ and $T_{\text{TLOTS}} \leq N - F$, so that the honest parties alone can make progress. Hence, our bulletin board is both consistent and robust only when $F \leq \lfloor \frac{N-1}{3} \rfloor$. At this maximal F , the bulletin-board threshold is approximately $2N/3$, whereas signing may use $T_{\text{TLOTS}} = F + 1 \approx N/3$; signing then requires few parties while consensus requires many, and an honest-but-curious adversary needs to corrupt relatively few parties to recover the secret. For $F = 0$, a strict majority suffices, i.e., $T_{\text{BB}} \geq \lfloor N/2 \rfloor + 1$.

The honest-but-curious model does not fit our scheme well, since deviating from the protocol *can* reveal extra secret information: disrupting consensus on the bulletin board enables a message-mismatch attack. An adversarial party is therefore incentivized to be more than honest-but-curious. If honest-but-curious signers are nevertheless a relevant threat, one can take $T_{\text{TLOTS}} \gg N/3$, which limits robustness but increases secrecy.

References

1. Agrawal, S., Stehlé, D., Yadav, A.: Round-Optimal Lattice-Based Threshold Signatures, Revisited. In: Bojańczyk, M., Merelli, E., Woodruff, D.P. (eds.) 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 229, pp. 8:1–8:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). <https://doi.org/10.4230/LIPIcs.ICALP.2022.8>, <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2022.8>
2. Aguilar Melchor, C., Albrecht, M.R., Bailleux, T., Bindel, N., Howe, J., Hülsing, A., Joseph, D., Manzano, M.: Batch signatures, revisited. In: Oswald, E. (ed.) CT-

- RSA 2024. LNCS, vol. 14643, pp. 163–186. Springer, Cham, Switzerland, San Francisco, CA, USA (May 6–9, 2024). https://doi.org/10.1007/978-3-031-58868-6_7
3. Alkeilani Alkadri, N., Döttling, N., Pu, S.: Practical lattice-based distributed signatures for a small number of signers. In: Pöpper, C., Batina, L. (eds.) ACNS 2024, Part I. LNCS, vol. 14583, pp. 376–402. Springer, Cham, Switzerland, Abu Dhabi, UAE (Mar 5–8, 2024). https://doi.org/10.1007/978-3-031-54770-6_15
 4. Bacho, R., Das, S., Loss, J., Ren, L.: Glacius: Threshold schnorr signatures from DDH with full adaptive security. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part II. LNCS, vol. 15602, pp. 304–334. Springer, Cham, Switzerland, Madrid, Spain (May 4–8, 2025). https://doi.org/10.1007/978-3-031-91124-8_11
 5. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS⁺ signature framework. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 2129–2146. ACM Press, London, UK (Nov 11–15, 2019). <https://doi.org/10.1145/3319535.3363229>
 6. Bienstock, A., de Castro, L., Escudero, D., Polychroniadou, A., Takahashi, A.: Quorus: Efficient, scalable threshold ML-DSA signatures from MPC. Cryptology ePrint Archive, Paper 2025/1163 (2025), <https://eprint.iacr.org/2025/1163>
 7. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 565–596. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 19–23, 2018). https://doi.org/10.1007/978-3-319-96884-1_19
 8. Borin, G., Celi, S., del Pino, R., Espitau, T., Katsumata, S., Niot, G., Prest, T., Takemure, K.: Hermine: An efficient lattice-based FROST-like threshold signature. In: Asiacrypt 2026 (to appear) (2026)
 9. Boschini, C., Kaviani, D., Lai, R.W.F., Malavolta, G., Takahashi, A., Tibouchi, M.: Ringtail: Practical two-round threshold signatures from learning with errors. In: Blanton, M., Enck, W., Nita-Rotaru, C. (eds.) IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12–15, 2025. pp. 149–164. IEEE (2025). <https://doi.org/10.1109/SP61157.2025.00070>, <https://doi.org/10.1109/SP61157.2025.00070>
 10. Brandão, L.T.A.N., Peralta, R.: NIST First Call for Multi-Party Threshold Schemes. NIST Interagency/Internal Report 8214C, National Institute of Standards and Technology (Jan 2026). <https://doi.org/10.6028/NIST.IR.8214C>, <https://doi.org/10.6028/NIST.IR.8214C>
 11. Bruinderink, L.G., Hülsing, A.: "oops, I did it again" - security of one-time signatures under two-message attacks. In: Adams, C., Camenisch, J. (eds.) Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16–18, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10719, pp. 299–322. Springer (2017). https://doi.org/10.1007/978-3-319-72565-9_15, https://doi.org/10.1007/978-3-319-72565-9_15
 12. Buchmann, J.A., Dahmen, E., Hülsing, A.: XMSS - A practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.Y. (ed.) Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011. pp. 117–129. Springer Berlin Heidelberg, Germany, Tapei, Taiwan (Nov 29 – Dec 2 2011). https://doi.org/10.1007/978-3-642-25405-5_8
 13. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (Oct 14–17, 2001). <https://doi.org/10.1109/SFCS.2001.959888>

14. Canetti, R., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part II. LNCS, vol. 9216, pp. 3–22. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 2015). https://doi.org/10.1007/978-3-662-48000-7_1
15. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1769–1787. ACM Press, Virtual Event, USA (Nov 9–13, 2020). <https://doi.org/10.1145/3372297.3423367>
16. Canetti, R., Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Adaptive security for threshold cryptosystems. In: Wiener, M.J. (ed.) CRYPTO’99. LNCS, vol. 1666, pp. 98–115. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 15–19, 1999). https://doi.org/10.1007/3-540-48405-1_7
17. Celi, S., del Pino, R., Espitau, T., Niot, G., Prest, T.: Efficient threshold ML-DSA. Cryptology ePrint Archive, Paper 2026/013 (2026), <https://eprint.iacr.org/2026/013>
18. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1825–1842. ACM Press, Dallas, TX, USA (Oct 31 – Nov 2, 2017). <https://doi.org/10.1145/3133956.3133997>
19. Chen, Y.: DualMS: Efficient lattice-based two-round multi-signature with trapdoor-free simulation. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part V. LNCS, vol. 14085, pp. 716–747. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 20–24, 2023). https://doi.org/10.1007/978-3-031-38554-4_23
20. Cozzo, D., Smart, N.P.: Sashimi: Cutting up CSI-FiSh secret keys to produce an actively secure distributed signing protocol. In: Ding, J., Tillich, J.P. (eds.) Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020. pp. 169–186. Springer, Cham, Switzerland, Paris, France (Apr 15–17, 2020). https://doi.org/10.1007/978-3-030-44223-1_10
21. Culnane, C., Schneider, S.A.: A peered bulletin board for robust use in verifiable voting systems. In: Datta, A., Fournet, C. (eds.) CSF 2014 Computer Security Foundations Symposium. pp. 169–183. IEEE Computer Society Press, Vienna, Austria (Jul 19–22, 2014). <https://doi.org/10.1109/CSF.2014.20>
22. Damgård, I., Orlandi, C., Takahashi, A., Tibouchi, M.: Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. *Journal of Cryptology* **35**(2), 14 (Apr 2022). <https://doi.org/10.1007/s00145-022-09425-3>
23. De Feo, L., Kohel, D., Leroux, A., Petit, C., Wesolowski, B.: SQISign: Compact post-quantum signatures from quaternions and isogenies. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part I. LNCS, vol. 12491, pp. 64–93. Springer, Cham, Switzerland, Daejeon, South Korea (Dec 7–11, 2020). https://doi.org/10.1007/978-3-030-64837-4_3
24. De Feo, L., Meyer, M.: Threshold schemes from isogeny assumptions. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) PKC 2020, Part II. LNCS, vol. 12111, pp. 187–212. Springer, Cham, Switzerland, Edinburgh, UK (May 4–7, 2020). https://doi.org/10.1007/978-3-030-45388-6_7
25. Del Pino, R., Katsumata, S., Maller, M., Mouhartem, F., Prest, T., Saarinen, M.J.O.: Threshold raccoon: Practical threshold signatures from standard lattice assumptions. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part II.

- LNCS, vol. 14652, pp. 219–248. Springer, Cham, Switzerland, Zurich, Switzerland (May 26–30, 2024). https://doi.org/10.1007/978-3-031-58723-8_8
26. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR TCHES* **2018**(1), 238–268 (2018). <https://doi.org/10.13154/tches.v2018.i1.238-268>, <https://tches.iacr.org/index.php/TCHES/article/view/839>
 27. Espitau, T., Katsumata, S., Takemure, K.: Two-round threshold signature from algebraic one-more learning with errors. In: Reyzin, L., Stebila, D. (eds.) *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference*, Santa Barbara, CA, USA, August 18–22, 2024, *Proceedings, Part VII*. pp. 387–424. *Lecture Notes in Computer Science*, Springer (2024). https://doi.org/10.1007/978-3-031-68394-7_13, https://doi.org/10.1007/978-3-031-68394-7_13
 28. Espitau, T., Niot, G., Prest, T.: Flood and submerse: Distributed key generation and robust threshold signature from lattices. In: Reyzin, L., Stebila, D. (eds.) *CRYPTO 2024, Part VII*. LNCS, vol. 14926, pp. 425–458. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 18–22, 2024). https://doi.org/10.1007/978-3-031-68394-7_14
 29. Feldman, P.: A practical scheme for non-interactive verifiable secret sharing. In: *28th Annual Symposium on Foundations of Computer Science*, Los Angeles, California, USA, 27–29 October 1987. pp. 427–437. IEEE Computer Society (1987). <https://doi.org/10.1109/SFCS.1987.4>
 30. Feo, L.D., Galbraith, S.D.: Seasign: Compact isogeny signatures from class group actions. In: Ishai, Y., Rijmen, V. (eds.) *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Darmstadt, Germany, May 19–23, 2019, *Proceedings, Part III*. *Lecture Notes in Computer Science*, vol. 11478, pp. 759–789. Springer (2019). https://doi.org/10.1007/978-3-030-17659-4_26, https://doi.org/10.1007/978-3-030-17659-4_26
 31. Frankel, Y., MacKenzie, P.D., Yung, M.: Adaptively secure distributed public-key systems. *Theor. Comput. Sci.* **287**(2), 535–561 (2002). [https://doi.org/10.1016/S0304-3975\(01\)00260-2](https://doi.org/10.1016/S0304-3975(01)00260-2)
 32. Gao, L., Tang, G., Jia, D., Liu, Y., Liu, B., Lu, X., Wang, K., Yin, Y.: Talong: Bandwidth-efficient two-round threshold signatures from lattices. *IACR Cryptol. ePrint Arch.* **2026**, 303 (2026), <https://eprint.iacr.org/2026/303>
 33. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) *ACM CCS 2018*. pp. 1179–1194. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). <https://doi.org/10.1145/3243734.3243859>
 34. Gür, K.D., Katz, J., Silde, T.: Two-round threshold lattice-based signatures from threshold homomorphic encryption. In: Saarinen, M.J., Smith-Tone, D. (eds.) *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Part II*. pp. 266–300. Springer, Cham, Switzerland, Oxford, UK (Jun 12–14, 2024). https://doi.org/10.1007/978-3-031-62746-0_12
 35. Hülsing, A.: WOTS+ – shorter signatures for hash-based signature schemes. *Cryptology ePrint Archive*, Report 2017/965 (2017), <https://eprint.iacr.org/2017/965>
 36. Kalach, K., Safavi-Naini, R.: An efficient post-quantum one-time signature scheme. In: Dunkelman, O., Keliher, L. (eds.) *SAC 2015*. LNCS, vol. 9566, pp. 331–351. Springer, Cham, Switzerland, Sackville, NB, Canada (Aug 12–14, 2016). https://doi.org/10.1007/978-3-319-31301-6_20

37. Katsumata, S., Reichle, M., Takemure, K.: Adaptively secure 5 round threshold signatures from MLWE/MSIS and DL with rewinding. In: Reyzin, L., Stebila, D. (eds.) CRYPTO 2024, Part VII. LNCS, vol. 14926, pp. 459–491. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 18–22, 2024). https://doi.org/10.1007/978-3-031-68394-7_15
38. Katz, J., Schröder, D., Yerukhimovich, A.: Impossibility of blind signatures from one-way permutations. In: Ishai, Y. (ed.) Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28–30, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6597, pp. 615–629. Springer (2011). https://doi.org/10.1007/978-3-642-19571-6_37, https://doi.org/10.1007/978-3-642-19571-6_37
39. Komlo, C., Goldberg, I.: FROST: Flexible round-optimized Schnorr threshold signatures. In: Dunkelman, O., Jacobson, Jr., M.J., O’Flynn, C. (eds.) SAC 2020. LNCS, vol. 12804, pp. 34–65. Springer, Cham, Switzerland, Halifax, NS, Canada (Virtual Event) (Oct 21–23, 2020). https://doi.org/10.1007/978-3-030-81652-0_2
40. Lamport, L.: Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International (1979)
41. Li, L., Xu, M.: Ripplesign: Isogeny-based threshold ring signatures with combinatorial methods. In: 6th International Conference on Cryptography, Security and Privacy, CSP 2022, Tianjin, China, January 14–16, 2022. pp. 11–15. IEEE (2022). <https://doi.org/10.1109/CSP55486.2022.00010>
42. Libert, B., Joye, M., Yung, M.: Born and raised distributively: fully distributed non-interactive adaptively-secure threshold signatures with short shares. In: Halldórsson, M.M., Dolev, S. (eds.) 33rd ACM PODC. pp. 303–312. ACM, Paris, France (Jul 15–18, 2014). <https://doi.org/10.1145/2611462.2611498>
43. McGrew, D.A., Curcio, M., Fluhrer, S.R.: Leighton-micali hash-based signatures. RFC **8554**, 1–61 (2019). <https://doi.org/10.17487/RFC8554>, <https://doi.org/10.17487/RFC8554>
44. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO’89. LNCS, vol. 435, pp. 218–238. Springer, New York, USA, Santa Barbara, CA, USA (Aug 20–24, 1990). https://doi.org/10.1007/0-387-34805-0_21
45. National Institute of Standards and Technology: Module-lattice-based digital signature standard. Federal Information Processing Standards Publication FIPS 204, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). <https://doi.org/10.6028/NIST.FIPS.204>, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.204.pdf>
46. National Institute of Standards and Technology: Stateless hash-based digital signature standard. Federal Information Processing Standards Publication FIPS 205, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). <https://doi.org/10.6028/NIST.FIPS.205>, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.205.pdf>
47. National Institute of Standards and Technology: Fft-over-ntru-lattice digital signature standard (fn-dsa). FIPS 206 (Draft), U.S. Department of Commerce, Washington, D.C. (2026)
48. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO’91. LNCS, vol. 576, pp. 129–140. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 11–15, 1992). https://doi.org/10.1007/3-540-46766-1_9
49. Pino, R.D., Katsumata, S., Niot, G., Reichle, M., Takemure, K.: Unmasking TRaccoon: A lattice-based threshold signature with an efficient identifiable abort protocol. In: Kalai, Y.T., Kamara, S.F. (eds.) CRYPTO 2025, Part VI. LNCS, vol. 16005,

- pp. 423–456. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 17–21, 2025). https://doi.org/10.1007/978-3-032-01887-8_14
50. Rompel, J.: One-way functions are necessary and sufficient for secure signatures. In: Ortiz, H. (ed.) Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13–17, 1990, Baltimore, Maryland, USA. pp. 387–394. ACM (1990). <https://doi.org/10.1145/100216.100269>, <https://doi.org/10.1145/100216.100269>
51. Ruffing, T., Ronge, V., Jin, E., Schneider-Bensch, J., Schröder, D.: ROAST: Robust asynchronous Schnorr threshold signatures. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 2551–2564. ACM Press, Los Angeles, CA, USA (Nov 7–11, 2022). <https://doi.org/10.1145/3548606.3560583>
52. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979). <https://doi.org/10.1145/359168.359176>, <https://doi.org/10.1145/359168.359176>
53. Shoup, V.: Practical threshold signatures. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 207–220. Springer Berlin Heidelberg, Germany, Bruges, Belgium (May 14–18, 2000). https://doi.org/10.1007/3-540-45539-6_15
54. Stadler, M.: Publicly verifiable secret sharing. In: Maurer, U.M. (ed.) EUROCRYPT’96. LNCS, vol. 1070, pp. 190–199. Springer Berlin Heidelberg, Germany, Saragossa, Spain (May 12–16, 1996). https://doi.org/10.1007/3-540-68339-9_17
55. Yuan, Q.: Improving Hash-Based Signature Schemes: From Theory to Practice, pp. 407–424 (10 2025). https://doi.org/10.1007/978-981-96-1218-5_21

A Components

A.1 Tweakable Hash Functions

Most hash-based structures use a *tweakable* hash function: an ordinary keyed hash function with one extra input, called *tweak*, that separates uses within one key, while the key itself separates different parties. Used correctly, this lets one tightly achieve target collision resistance even in *multi-target* settings, where an adversary wins by breaking just one of many targets.

Definition 3 (Tweakable Hash Function [5]) Let $\alpha \in \mathbb{N}$, $\tilde{\mathcal{P}}$ a public parameter space, and $\mathcal{T}$ a tweak space. A tweakable hash function is an efficient function

$$\mathbf{H} : \tilde{\mathcal{P}} \times \mathcal{T} \times \{0, 1\}^\alpha \rightarrow \{0, 1\}^\lambda, \quad h \leftarrow \mathbf{H}(P, t, x),$$

mapping an α -bit message x under a public parameter $P \in \tilde{\mathcal{P}}$ and tweak $t \in \mathcal{T}$ to a λ -bit digest h .

We review two targeted security properties of $\mathbf{H}$: multi-target collision resistance, and decisional second-preimage resistance.

Definition 4 (pq-sm-tcr [5]) Let $\mathbf{H}$ be a tweakable hash function, $q \leq |\mathcal{T}|$ a target count, and $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ an adversary: $\mathcal{A}_1$ adaptively queries an oracle $\mathbf{H}(P, \cdot, \cdot)$ up to q times, specifying q targets; write $\mathcal{Q} = \{(t_i, x_i)\}_{i=1}^q$ for its queries, and $\text{DIST}(\{t_i\}_{i=1}^q) := [\forall i, k \in [1, q], i \neq k : t_i \neq t_k]$, i.e. $\text{DIST}(\{t_i\}_{i=1}^q)$ outputs 1 iff all tweaks are distinct. Define

$$\text{Succ}_{\mathbf{H}, q}^{\text{sm-tcr}}(\mathcal{A}) := \Pr \left[P \xleftarrow{\$} \tilde{\mathcal{P}}; S \leftarrow \mathcal{A}_1^{\mathbf{H}(P, \cdot, \cdot)}(); (j, x) \leftarrow \mathcal{A}_2(\mathcal{Q}, S, P) : \right.$$

$$H(P, t_j, x_j) = H(P, t_j, x) \wedge x \neq x_j \wedge \text{DIST}(\{t_i\}_{i=1}^q)].$$

The pq -sm-tcr insecurity of H against q -target, time- ξ adversaries is $\text{InSec}^{\text{pq-sm-tcr}}(H; \xi, q) := \max_{\mathcal{A}} \{\text{Succ}_{H,q}^{\text{sm-tcr}}(\mathcal{A})\}$, over (possibly quantum) $\mathcal{A}$ with q targets and running time $\leq \xi$.

For a tweakable hash function H , the second-preimage-exists predicate is $\text{SP} : \tilde{\mathcal{P}} \times \mathcal{T} \times \{0, 1\}^\alpha \rightarrow \{0, 1\}$,

$$\text{SP}_P(t, x) := \begin{cases} 1 & \text{if } |H_{P,t}^{-1}(H_{P,t}(x))| \geq 2, \\ 0 & \text{otherwise.} \end{cases}$$

where $H_{P,t}^{-1}$ is the inverse of H with its first two inputs fixed to P, t .

Definition 5 (pq-sm-dspr [5]) Under the same setup as Definition 4, define

$$\begin{aligned} \text{succ} &:= \Pr \left[P \xleftarrow{\$} \tilde{\mathcal{P}}; S \leftarrow \mathcal{A}_1^{H(P, \cdot)}(); (j, b) \leftarrow \mathcal{A}_2(\mathcal{Q}, S, P) : \text{SP}_{P,t_j}(x_j) = b \wedge \text{DIST}(\{t_i\}_{i=1}^q) \right], \\ \text{triv} &:= \Pr \left[P \xleftarrow{\$} \tilde{\mathcal{P}}; S \leftarrow \mathcal{A}_1^{H(P, \cdot)}(); (j, b) \leftarrow \mathcal{A}_2(\mathcal{Q}, S, P) : \text{SP}_{P,t_j}(x_j) = 1 \wedge \text{DIST}(\{t_i\}_{i=1}^q) \right]. \end{aligned}$$

and $\text{Adv}_{H,P}^{\text{sm-dspr}}(\mathcal{A}) := \max\{0, \text{succ} - \text{triv}\}$ – the advantage over the trivial adversary that always guesses $b = 1$. The pq -sm-dspr insecurity of H against q -target, time- ξ adversaries is $\text{InSec}^{\text{pq-sm-dspr}}(H; \xi, q) := \max_{\mathcal{A}} \{\text{Adv}_{H,P}^{\text{sm-dspr}}(\mathcal{A})\}$, over (possibly quantum) $\mathcal{A}$ with q targets and running time $\leq \xi$.

A.2 Winternitz one-time signature scheme.

WOTS is designed for lightweight signing and verification. WOTS^+ , one of the variants of WOTS, is used as a core component in standardized hash-based signature schemes such as XMSS and SPHINCS⁺. WOTS^+ relies on two key parameters: the security parameter λ and the Winternitz parameter w . These parameters are used to define

$$\ell_1 = \lceil \frac{\lambda}{w} \rceil, \quad \ell_2 = \lfloor \frac{\log(\ell_1(2^w - 1))}{w} \rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

Let $\lambda \in \mathbb{N}$ be a security parameter, $H : \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ be a tweakable hash function, and prm_{WOTS} and prm_{msg} be public parameters. Each hash call in a chain uses a tweak that also encodes the chain index i and the call's height j in the chain, i.e., $t\|i\|j$ (with all components encoded with fixed length), so no two hash calls share a tweak. We write $H^k(P, t\|i\|j, x)$ for k chain steps applied to x starting at height j :

$$H^0(P, t\|i\|j, x) = x, \quad H^{k+1}(P, t\|i\|j, x) = H(P, t\|i\|(j+k), H^k(P, t\|i\|j, x)).$$

Then

$$\text{WOTS}^+ = (\text{WKeyGen}, \text{WSign}, \text{WDerivePK}, \text{WVerify}),$$

an instance of Definition 1, is defined as:

- $(\text{sk}^W, \text{pk}^W) \leftarrow \text{WKeyGen}(1^\lambda, t)$: Samples $x_i \xleftarrow{\$} \{0, 1\}^\lambda$ for each $0 \leq i \leq \ell - 1$, and sets $\text{sk}^W = (x_0, \dots, x_{\ell-1})$, $\text{pk}^W = (y_0, \dots, y_{\ell-1})$ with $y_i = H^{2^w-1}(\text{prm}_{\text{WOTS}}, t \| i \| 0, x_i)$.
- $\sigma^W \leftarrow \text{WSign}(\text{sk}^W, \text{msg}, t)$: Computes $D = H(\text{prm}_{\text{msg}}, t, \text{msg}) = (D_0, \dots, D_{\ell_1-1})_{2^w}$ and the checksum $C = \sum_{i=0}^{\ell_1-1} (2^w - 1 - D_i) = (D_{\ell_1}, \dots, D_{\ell-1})_{2^w}$, and outputs

$$\sigma^W = (\sigma_0, \dots, \sigma_{\ell-1}), \quad \sigma_i = H^{D_i}(\text{prm}_{\text{WOTS}}, t \| i \| 0, x_i) \text{ for all } 0 \leq i \leq \ell - 1.$$

- $\text{pk}^{W'} \leftarrow \text{WDerivePK}(\text{msg}, \sigma^W, \text{pk}^{\text{cmpl}}, t)$: Computes $(D_0, \dots, D_{\ell-1})_{2^w}$ from msg exactly as in WSign , ignores $\text{pk}^{\text{cmpl}} = \emptyset$ since every component of pk^W is recoverable from (msg, σ^W) alone, and outputs the candidate public key

$$\text{pk}^{W'} = (H^{2^w-1-D_0}(\text{prm}_{\text{WOTS}}, t \| 0 \| D_0, \sigma_0), \dots, H^{2^w-1-D_{\ell-1}}(\text{prm}_{\text{WOTS}}, t \| (\ell-1) \| D_{\ell-1}, \sigma_{\ell-1})).$$

- $0/1 \leftarrow \text{WVerify}(\text{pk}^W, \text{msg}, \sigma^W, t)$: Compute $\text{pk}^{W'}$ as in WDerivePK and accepts the signature σ^W iff $\text{pk}^{W'} = \text{pk}^W$.

A.3 Merkle Trees

We detail here the `MerkleTree` syntax introduced in Section 2.2, following the tweaked and domain-separated structure from [2]. Following their notation (from [5]), we let H be a tweakable hash function: it has input $H(P, t, m)$, where P is a public parameter, t is a tweak, and m is the message.

A Merkle tree MT of height h built by $\text{MerkleTree}(\text{id}_{\text{MT}}, x_0, \dots, x_{2^h-1})$ has 2^h leaves. We denote by $y_j[i]$ the i th node at level j , where level 0 is the root level and level h is the leaf level, and set $y_h[i] = H(\text{id}_{\text{MT}}, 0 \| i, x_i)$. For a *salted* Merkle tree, for each input x_i we sample a random $r_i \in \{0, 1\}^\lambda$ and set $y_h[i] = H(\text{id}_{\text{MT}}, 0 \| i, r_i \| x_i)$. Internal nodes are then computed bottom-up by

$$y_j[i] = H(\text{id}_{\text{MT}}, 1 \| j \| i, y_{j+1}[2i] \| y_{j+1}[2i+1]), \quad 0 \leq j < h, \quad 0 \leq i < 2^j,$$

and MT.Root returns $y_0[0]$, a compact commitment to all 2^h leaves. $\text{MT.Path}(i)$ returns the authentication path $\text{path}_i = (a_1, \dots, a_h)$, i.e., the sibling of each node on the path from $y_h[i]$ to the root. $\text{MerkleTree.DeriveRoot}(x_i, \text{path}_i)$ recomputes a candidate root from $y_h[i] = H(\text{id}_{\text{MT}}, 0 \| i, x_i)$, the index i , and path_i , hashing each node with its sibling in the correct left-right order; a claim that x_i is the i th leaf of MT is accepted iff this candidate root equals MT.Root . Figure 6 shows a Merkle tree of height 4, highlighting the authentication path (yellow) and the computed nodes (green) required to verify the leaf node $y_4[7]$.

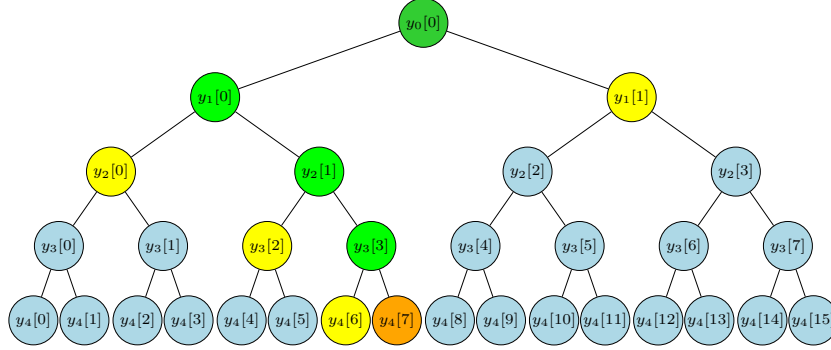


Fig. 6: A Merkle tree of height 4.

B Bulletin Board

Real Protocol. We realize the bulletin board as a permissioned blockchain, by designating an aggregator, and having the other parties provide a majority vote on each new message, which is given a “block”. Each “vote” is signature from a long-term signature scheme $\text{SIG} = (\text{SIG.KeyGen}, \text{SIG.Sign}, \text{SIG.Verify})$. A block B contains a message $B[[m]]$, a hash of the previous block $B[[h]]$, the sender $B[[\text{sender}]]$, and at least $\frac{N+F+1}{2}$ signatures $B[[\text{signs}]]$. This ensures that at least half of the honest participants have signed any block. As we explained in Section 6, if each honest party ensures they do not sign two different blocks at the same index in the chain, then only one block can reach the required number of signatures.

This relies on an initial trusted key distribution step. Any method would do, but since we have already defined an ideal key generation functionality, we will use this. This ideal functionality is parameterized by a key generation algorithm and a share issuing algorithm, which we specify as BB.KeyGen . Overall this provides an initial consensus on public keys, and any public key infrastructure would suffice. Since our threshold signature scheme expects a trusted third party to distribute key shares, we could have this entity also distribute these public keys during the setup phase.

The aggregator is responsible for constructing new blocks and requesting signatures from the other parties. A malicious aggregator cannot force inconsistency in the bulletin board, but can halt progress indefinitely by either refusing to send any messages, or sending inconsistent blocks to honest users. The honest users will never accept the inconsistent blocks as they will not have enough signatures, but we provide no mechanism to recover from such inconsistency. This means our aggregator can be malicious as far as unforgeability or identifiable abort of the signature scheme, but must be trusted for robustness. We give an overview of the function of these algorithms:

BB.Activate: This establishes the bulletin board and calls the key generation $\mathcal{F}_{\text{DKG}}^{\text{BB.KeyDist}}$, which will distribute the public keys necessary for consensus. As

in TLOTS, we are again calling an ideal key generation as a subroutine of the real protocol.

BB.Post: When requested to post, all users except the aggregator $\mathcal{M}$ simply forward the request to the aggregator. The aggregator maintains a queue of ongoing requests; if the aggregator is still waiting for signatures from a previous request (denoted by `state.inProgress`) then they simply add the queue and stop. Otherwise, they can immediately create a new block and share it, via the helper function **BB.CreateAndShare**.

BB.Read: This algorithm can be run by any machine, not necessarily hosts. This ensures that any machine can consistently interact with the bulletin board, which is hosted by several machines, and avoids the problem of asking a single adversarial host for the state of the bulletin board and receiving an arbitrary response.

Of course, if the requestor *is* a host, they can simply inspect their own copy of the bulletin board and return the response, which is useful for TLOTS, where all users of the bulletin board are also hosts.

In any other case, one sends **BB.Query** messages to all of the hosts, and simply waits for a majority response once enough participants respond.

BB.Search: This is primarily a convenience function, so we simply wrap **BB.Read**. This function does not halt until it finds an index; if it reaches the end of all posted messages it will simply wait.

Algorithm 7 Externally accessible **BB** algorithms. These can be called by any party.

BB.Activate(sid, $\mathcal{P}$, F)

- 1: Initialize $B = []$, $\text{corrupt} = \emptyset$, save $B, \text{corrupt}, \mathcal{P}, F$ to state.
- 2: Send (sid, $F, N - F, \infty, \mathcal{P}$) to $\mathcal{F}_{\text{DKG}}^{\text{BB.KeyDist}}$
- 3: Send (**keygen**, sid) to $\mathcal{F}_{\text{DKG}}^{\text{BB.KeyDist}}$
- 4: Broadcast (**BB.Activate**, sid, $\mathcal{P}, F$) to all $P_i \in \mathcal{P}$ if state.initialized is False
- 5: state.initialized $\leftarrow$ True
- 6: state.queries $\leftarrow \emptyset$

BB.Post(msg, senderID):

- 1: **if** $P_{\text{id}} \neq \mathcal{M}$ **then** $\triangleright \mathcal{M}$ stands for the aggregator
- 2: Forward (**BB.Post**, msg, senderID) to state. $\mathcal{M}$
- 3: Return
- 4: **else**
- 5: **if** state.inProgress $\neq \perp$ **then**
- 6: Append (msg, senderID) to the end of state.signingQueue
- 7: Return
- 8: **end if**
- 9: **BB.CreateAndShare**(msg, senderID)
- 10: **end if**

BB.Read(i , senderID):

- 1: **if** $P_{\text{id}} \in \mathcal{P}$ and state. $B[i] \neq \perp$ **then** Return state. $B[i][[m]]$ to senderID
- 2: **else**
- 3: **for** $P_j \in \mathcal{P}$ **do**
- 4: Send (**BB.Query**, i) to P_j
- 5: **end for**
- 6: **end if**
- 7: Return the majority answer from (**response**, msg) messages from $P_j \in \mathcal{P}$ once least $\frac{N+F+1}{2}$ responses are received and a majority is reached. If every $P_j \in \mathcal{P}$ returns a response but no majority answer is reached, return $\perp$.

BB.Search(msg, senderID):

- 1: **for** $i = 0$ until returning **do**
- 2: **if** **BB.Read**(i) = msg **then** Return i to senderID
- 3: **end if**
- 4: **end for**

B.1 Internal Functions

As with TLOTS, we have internal functions which we describe here.

BB.ReceiveKeys: Called by $\mathcal{F}_{\text{DKG}}^{\text{BB.KeyDist}}$ when it sends the long-term signing keys for the bulletin board. The host, who receives them, simply records the values.

Algorithm 8 **BB.KeyDist**: Key generation algorithms for the bulletin board. Simply generates a long-term signing keypair for each users; more importantly, it gives consensus on their public keys. The “KeyRefresh” algorithm is omitted as it would do nothing.

```

  BB.KeyGen( $F, T, \mathcal{P}, M = \infty$ )
1: for  $P_i \in \mathcal{P}$  do
2:    $(\text{sk}_i, \text{pk}_i) \leftarrow \text{SIG.KeyGen}(1^\lambda) \triangleright (\text{sk}_i, \text{pk}_i)$  is the long-term signing key pair of  $P_i$ 
3: end for
4: Return  $(\text{sk} = \{\text{sk}_i\}_{i \in [N]}, \text{pk} = \{\text{pk}_i\}_{i \in [N]})$ 

```

```

  BB.IssueShares( $\text{sk}, T, N, p, \{(i, \text{sk}_i)\}_{i \in \mathcal{C}}$ )
1: Return  $\{\text{sk}_i\}_{i \in \mathcal{P}}$ 

```

BB.Query: This is called by a user attempting to read from the bulletin board. If this host’s local bulletin board has not reached index i , the host saves the query; it will check with each new block whether it can respond. If it has an i th block, it returns it.

BB.Sign: Mainly this wraps **BB.SignInternal**, which handles the more complicated logic of validating a block before signing. This function primarily checks whether this user has already been requested to sign this block, and that the request came from the aggregator, and maintains a list of requests. Maintaining the list ensures that if signing requests appear out-of-order, honest hosts can eventually catch up when messages are finally delivered.

BB.Status: Informs other users of how far each user’s blockchain is. Each host maintains an internal status **otherStates** of how far it thinks each other host has progressed in the blockchain, so this allows them to update this status. If the recipient of a **BB.Status** message has more blocks, they return them to the sender.

BB.AggregateBlock: The aggregator runs this only, in response to a signature on a block. The aggregator checks that the signature is valid, and updates its record of how far the sender has progressed in the chain and saves the signature. Once it has enough signatures (a supermajority, effectively), it finalizes this block and adds it to its own blockchain. It calls **BB.Propagate**, which sends this new block to all other users.

It then checks whether it had other messages queued to post in its **signQueue**. If so, it restarts the process by calling **BB.CreateAndShare**.

BB.AddBlocks: This does a majority of the logic. The input is a list of new blocks, and the index where they should start. If the index is already past the last block that the recipient has, they return a **BB.Status** request and stop, which should trigger the sender to send them a longer list of missing blocks.

The opposite case is where the recipient has so many blocks that the new ones would not give any new information. In this case, they send their own extension of the blockchain back to the sender.

In any other case, the recipient validates and adds the new blocks. They ignore any blocks they might already have (we could do a consistency check to catch bad behaviour, but if the new blocks are valid, it doesn't matter), validate each block and add it to their blockchain. With each new block, there is a possibility that they can now respond to a previous "Read" query, so they iterate through their saved queries and respond to any they can. Finally, this host will check whether it can respond to any previous signing requests.

B.2 Helper Functions

Below are functions that cannot be called by an external message, but are only run internally as helper functions. We describe them below:

- BB.NewBlock:** Given a new message, it formats a new block, including a hash of the previous block.
- BB.Propagate:** Sends out as many new blocks as possible. As mentioned, each user tracks how many blocks it thinks each other user has, so it can try to fill in the gaps with this function.
- BB.ValidateBlock:** This ensures that a block is valid: that it contains the hash of the previous block, and that it has a supermajority of valid signatures. We choose to ignore invalid signatures here, rather than attempting an aggressive "blaming" procedure.
- BB.SignInternal:** This iterates through all queued signature requests and checks if they can be signed. Since it is pointless to sign a previous block in the chain (it must already have enough signatures), this only checks if there are signatures that would go on the next block for this host. It checks again that it has not signed anything for this index before, and checks that the new block contains a hash of the last one, and then it signs and makes a record of what it signed. Notice that if it has signed the same block before, it will sign it again; this will not harm correctness, but is easier to describe and could better handle packet loss.
- BB.CreateAndShare:** This is run by the aggregator. It generates a new block, saves it as the `inProgress` block, and adds its own signature. Before sending it out to be signed by all hosts, the aggregator attempts to catch up all the hosts by calling **BB.Propagate**.

Algorithm 9 Internal BB algorithms. These are not callable by parties outside of $\mathcal{P}$ (i.e., each function implicitly returns immediately if $\text{senderID} \notin \mathcal{P}$)

```

BB.ReceiveKeys(sid, pk, sk)
1: // Sent from  $\mathcal{F}_{\text{DKG}}^{\text{BB, KeyDist}}$ ; pk is a list of public keys for each host
2: Add pk and sk to state

```

```

BB.Query( $i$ , senderID)
1: msg  $\leftarrow$  state. $B[i][m]$ 
2: if msg not found then Add ( $i$ , senderID) to state.queries
3: else Send (response, msg) to senderID
4: end if

```

```

BB.Sign( $i$ , Blocknew, senderID)
1: if state.signed[ $i$ ] =  $\perp$  then
2:   if state.signRequests[ $i$ ] already exists or senderID  $\neq$  state. $\mathcal{M}$  then Return
3:   end if
4:   Add ( $i$  : Blocknew) to state.signRequests
5: end if
6: BB.SignInternal()

```

```

BB.Status( $i$ , senderID)  $\triangleright$  Responds to requests from other users with more blocks.
1: state.otherStates[senderID]  $\leftarrow i$ 
2: if length(state. $B$ ) >  $i$  then
3:   Send (BB.AddBlocks,  $i$ , state. $B[i : \text{end}]$ , Blocknew =  $\perp$ )
4: end if

```

```

BB.AggregateBlock( $i$ ,  $\Sigma^*$ , senderID)  $\triangleright$  Run by the aggregator when it receives a signature
1: if  $P_d \neq \mathcal{M}$  then return  $\triangleright$  Only the aggregator should run this
2: end if
3: if SIG.Verify(state.pk[senderID], state.inProgress,  $\Sigma^*$ ) returns False then Return
4: end if
5: state.otherStates[senderID]  $\leftarrow i - 1$   $\triangleright$  Aggregator learned this sender's blockchain has length  $i - 1$ 
6: state.nextSigs[senderID]  $\leftarrow \Sigma^*$ 
7:  $N \leftarrow |\text{state}.\mathcal{P}|$ ;  $F \leftarrow \text{state}.F$ 
8: if state.nextSigs has at least  $\frac{N+F+1}{2}$  valid signatures then
9:   Add state.nextSigs to state.inProgress
10:   Append state.inProgress to  $B$ 
11:   state.inProgress  $\leftarrow \perp$ 
12:   BB.Propagate()  $\triangleright$  Send out new block(s)
13:   if state.signQueue is non-empty then
14:     (msg, senderID)  $\leftarrow$  pop the first element from state.signQueue
15:     BB.CreateAndShare(msg, senderID)  $\triangleright$  Requests signatures for next message to add to
        blockchain, if it exists
16:   end if
17: end if

```

```

BB.AddBlocks(index, missingBlocks, senderID)
// If the new blocks are not contiguous with the existing blockchain, request more blocks
1: if index > length(state. $B$ ) then
2:   Send (BB.Status, length(state. $B$ )) to senderID
3:   Return
4: end if
// If the blocks overlap with the existing blockchain, ensure that they match
5: for  $i = \text{index}$  to length(state. $B$ ) do
6:   if missingBlocks[ $i - \text{index}$ ]  $\neq$  state. $B[i]$  then Return
7: end if
8: end for
9: if index + length(missingBlocks) < length(state. $B$ ) then  $\triangleright$  User has more blocks than they were sent
10:   Send (BB.AddBlocks, index, state. $B[\text{index} : \text{end}]$ ) to  $i$   $\triangleright$  Help sender expand their blockchain
11: end if
// Here we validate and incorporate new blocks
12: for  $i$  from length(state. $B$ ) - index to length(missingBlocks) do
13:   Blocknext  $\leftarrow$  (missingBlocks[ $i$ ][ $h$ ], missingBlocks[ $i$ ][msg], missingBlocks[ $i$ ][sender])
14:   if BB.ValidateBlock(missingBlocks[ $i$ ] - 1,  $i$  - 1, Blocknext) returns False then Return
15:   end if
16:   Append missingBlocks[ $i$ ] to state. $B$ 
17:    $\triangleright$  Each new block might permit responding to an earlier "Read" query
18:   for ( $j$ , senderID') in state.queries such that  $j = i + \text{index}$  do
19:     Send (response, missingBlocks[msg]) to senderID
20:     Remove ( $j$ , senderID') from state.queries
21:   end for
22: end for
23:  $\mu \leftarrow \text{length}(B)$ 
24: state.otherStates[senderID]  $\leftarrow \mu$   $\triangleright$  We now know that senderID has blocks up to  $\mu$ 
25: BB.SignInternal()  $\triangleright$  With new blocks, this user might now be able to respond to previous
    signing queries

```

Algorithm 10 “Helper” functions for BB that cannot be called by external messages, but only as subroutines of other functions.

BB.NewBlock($B, (\text{msg}, \text{senderID})$)

- 1: $\text{Block}_{\text{new}}[[h]] \leftarrow H_{\text{BB}}(|B|, B[\text{last}])$
- 2: $\text{Block}_{\text{new}}[[m]] \leftarrow \text{msg}$
- 3: $\text{Block}_{\text{new}}[[\text{sender}]] \leftarrow \text{senderID}$
- 4: Return $\text{Block}_{\text{new}}$

BB.Propagate()

- 1: **for** $P_i \in \mathcal{P} \setminus \{P_{\text{id}}\}$ **do**
- 2: $\text{missingBlocks} \leftarrow B[\text{state.otherStates}[i]]$
- 3: Send (**BB.AddBlocks**, $\text{state.otherState}[i]$, missingBlocks) to P_i
- 4: **end for**

BB.ValidateBlock($\text{Block}_{\text{prev}}, \text{index}, \text{sigs}, \text{Block}_{\text{next}}$) ▷ sigs is the set of long signature of the block

- 1: **if** $\text{Block}_{\text{next}}[[h]] \neq H_{\text{BB}}(\text{index}, \text{Block}_{\text{prev}})$ **then** Return False
- 2: **end if**
- 3: **if** $|\{i \in \{P_1, \dots, P_N\} \mid \text{SIG.Verify}(\text{state.pk}[i], \text{Block}_{\text{next}}, \Sigma_i^*) = 1\}| < \frac{N+F+1}{2}|$ **then** return False
- 4: **end if**
- 5: Return True

BB.SignInternal()

- 1: $\mu \leftarrow \text{length}(\text{state.B}) + 1$
- 2: Select any message from $\text{state.signRequests}[\mu]$
- 3: (If no such message exists, return)
- 4: Parse the message as $(\mu, \text{Block}_{\text{new}})$
- 5: **if** $\text{state.signed}[\mu] \notin \{\perp, \text{Block}_{\text{new}}\}$ or $\text{Block}_{\text{new}}[[h]] \neq H(B[\mu - 1])$ **then** Return
- 6: **end if**
- 7: $\Sigma^* \leftarrow \text{SIG.Sign}(\text{state.sk}, \text{Block}_{\text{new}})$
- 8: $\text{state.signed}[\mu] \leftarrow \text{Block}_{\text{new}}$
- 9: Send (**BB.AggregateBlock**, μ, Σ^*) to state.M

BB.CreateAndShare($\text{msg}, \text{senderID}$)

- 1: $\text{Block}_{\text{new}} \leftarrow \text{BB.NewBlock}(\text{state.B}, (\text{msg}, \text{senderID}))$
- 2: $\text{state.inProgress} \leftarrow \text{Block}_{\text{new}}$
- 3: $\Sigma_{\text{id}}^* \leftarrow \text{SIG.Sign}(\text{state.sk}, \text{Block}_{\text{new}})$
- 4: $\text{state.nextSigs}[\text{id}] \leftarrow \Sigma_{\text{id}}^*$
- 5: **BB.Propagate**()
- 6: Broadcast (**BB.Sign**, $\text{length}(\text{state.B})$, $\text{Block}_{\text{new}}$)

C Security Proof of Theorem 1

Theorem 1. *TLOTS UC-securely realizes the ideal functionality $\mathcal{F}_{\text{Sig}}$. That is, for any adversary **Adv** against TLOTS there exists a simulator **Sim** such that for any environment UC.Env, the advantage of UC.Env in distinguishing the real and ideal executions is bounded by $9\text{Adv}^{\text{SM-TCR}}(\text{H}) + 2\text{Adv}^{\text{SM-DSPR}}(\text{H}) + \text{Adv}^{\text{eUF-CMA}}(\text{B})$.*

Proof. We proceed through a series of hybrid games. We will build the ideal functionality piece by piece. Thus, for honest parties, we assume that when given a command message for the ideal functionality (e.g. **sign**), they will forward it to **Sim** if that command is not yet part of the ideal functionality, but will forward to the ideal functionality otherwise.

In the “zeroth” game, the simulator will internally run the adversary and play the role of the honest parties exactly as specified in the real protocol. This is identical to the real protocol, except that the simulator now has access to all of the adversary’s communication (which it forwards unchanged to UC.Env as needed).

First Game: The simulator does not forward any messages between the adversary and the ideal key generation, and instead runs the key generation itself, precisely as the ideal functionality does. The environment has no direct access to the ideal key generation functionality, so this is perfectly indistinguishable from the 0th game. However, this gives the simulator access to the secret shares of all corrupt and honest players.

Second Game: We introduce a partial ideal functionality $\mathcal{F}_1$ that only runs the Corruption subroutine of $\mathcal{F}_{\text{Sig}}$. On receiving a corruption request from **Adv** for player j , **Sim** checks whether j is compromised in $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$; if not, **Sim** does nothing, and otherwise sends (**corrupt**, P_j) to $\mathcal{F}_1$ and sends the internal state of j to **Adv**. This is a perfect simulation.

Third Game: There is a new partial ideal functionality $\mathcal{F}_2$ that adds the Initialization and Key Generation algorithms to $\mathcal{F}_1$. When the environment sends (**keygen**, sid) to any honest user, this message is forwarded without modification by **Sim** to $\mathcal{F}_2$.

When **Sim** receives (**initialize**, $\text{sid}, F, T, M, \mathcal{P}$) from the ideal functionality, it will run $\text{TLOTS.Initialize}(\text{sid}, F, T, M, \mathcal{P})$ for all honest parties. This in turn will trigger it running its local version of $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}$.

When **Sim** receives (**keygen**, sid, i) from $\mathcal{F}_2$ to an honest player i , it will run $\text{TLOTS.KeyGen}(\text{sid}, F, T, \mathcal{P})$ for its internal honest player P_i . Since this will call $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}.\text{keygen}$ (which is now run by **Sim**), **Sim** will follow the protocol and send $(\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}.\text{keygen}, i, \text{sid})$ to **Adv**. If **Sim** receives such a message for corrupt i , it will just forward $(\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}.\text{keygen}, i, \text{sid})$ to **Adv**.

When **Sim** receives $(\mathcal{C}, \text{sid}, \{\text{sk}_i\}_{i \in \mathcal{C}})$ from **Adv**, it will run $\mathcal{F}_{\text{DKG}}^{\text{TLOTS.KeyDist}}.\text{s}$ key generation as expected and forward the required messages. If this stage produces no key, **Sim** will forward nothing to its own ideal functionality $\mathcal{F}_2$.

Otherwise, **Sim** will send **(initialize, $\mathcal{V}$, pk)** to $\mathcal{F}_2$, where $\mathcal{V} = \text{TLOTS.Verify}$. **Sim** will deliver the **pubkey** messages returned by $\mathcal{F}_2$ to the real honest users exactly when **Adv** returns the **(pk, sk_i)** messages to the simulated honest users.

This is a perfect simulation as well, as the simulator effectively forwards all messages as expected. In the real protocol, honest users output **(pubkey, pk^B, sid)** at the end of the key generation stage, and in this hybrid game **Sim** sends **initialize** to $\mathcal{F}_2$ at the end of the key generation phase, which then sends **(pubkey, pk^B, sid)** to its “dummy” honest users as well, who simply copy this to their output tape.

Fourth Game: In the fourth game, **Sim** halts if any two honest players within its simulation generate a value in their **signatures** array with the same **sigid** but a different value for **index** or **msg**. Obviously this would be trivial for the environment to distinguish, so we prove instead that this event does not occur.

We see that the value of **index** is set during **TLOTS.Sign** as the minimum index in the bulletin board for a “post” of the form **(sigid, msg)**. The bulletin board is append-only and returns the same index to all users; therefore, the minimum index will be the same for all honest users. This is the only time **index** is set.

Thanks to this hybrid, from now on we can unambiguously talk about the index associated to a given **sigid**.

Fifth game: Here **Sim** halts if, at any point in execution, it obtains a value σ which is a valid TLOTS signature $(\sigma^L, \text{pk}^{\text{cmpl}}, \text{index}, \text{Sig})$ such that **Sig** was not output during the key generation process as a valid signature of the public key derived from $\text{LDerivePK}(\text{prm}, \text{msg}, \sigma^L, \text{pk}^{\text{cmpl}}, \text{index})$. There is an easy hybrid adversary against the existential unforgeability of **B** which requests M signatures from $\mathcal{F}_{\text{Sig}}$ during each key generation phase of **TLOTS**, then runs **Sim** as above, and outputs **Sig** if such a valid signature is produced. Thus, we lose only a factor of $\text{Adv}^{\text{eUF-CMA}}(\text{B})$ in advantage from this game hop.

Sixth Game: In the sixth game, **Sim** may halt during verification. More precisely, suppose an honest user j^* (run by **Sim**) receives a message **(TLOTS.SignFailed, sigid, π)** from corrupt player j . If the honest party runs this without outputting a “failure” message, then the simulator does the following:

1. Let **index** be the message index the honest party obtained from its signatures dictionary
2. If $\pi[k] \neq \text{sh}_{\text{index}, j, k}$ for any $k \in B_{\text{msg}}$, the simulator halts;

Again, this is distinguishable if the simulator halts, so we prove that this situation implies a second-preimage for **H**. We know that the user was given $\text{vk}[j][\text{index}] = H_{\text{vk}}(\text{index} \| j, (H_{\text{vk}}(\text{index} \| j \| k, \text{sh}_{\text{index}, j, k}))_{k=1}^e)$ during signature generation. During signature verification, the user computes $H_{\text{vk}}(\text{index} \| j, (\pi'[1], \dots, \pi'[e]))$ (where $\pi'[k] = H_{\text{vk}}(\text{index} \| j \| k, \pi[k])$ for $k \in B_{\text{msg}}$ and $\pi'[k] = \pi[k]$ otherwise) and compares to $\text{vk}[j][\text{index}]$. We can construct a hybrid game against **SM – TCR** by using queries to the tweakable

hash function to run key generation for **Sim**. The above gives us a collision in one of two ways: (a) $H_{vk}(\text{index}||j||k, \pi[k]) = H_{vk}(\text{index}||j||k, \text{sh}_{\text{index},j,k})$, or $H_{vk}(\text{index}||j, (H_{vk}(\text{index}||j||k, \text{sh}_{\text{index},j,k}))_{k=1}^e) = H_{vk}(\text{index}||j, (\pi'[1], \dots, \pi'[e]))$. The tweaks are the same, but the messages are distinct unless $\pi[k] = \text{sh}_{\text{index},j,k}$.

Seventh Game: In the seventh game, **Sim** checks every $(\text{TLOTS.Sign}_2, \text{sigid}, \sigma_{\text{index},j}^L)$ message from an adversarial player j , and marks (j, sigid) as “malformed” if $\sigma_{\text{index},j}^L$ differs from the partial signature that an honest execution of **TLOTS.PartialSignature** would output on the shares of player j at the index for that **sigid** (recall that **Sim** holds all corrupted secret shares).

If an honest user receives $(\text{TLOTS.SignFailed}, \text{sigid}, \pi)$ from a player j for which **Sim** has recorded (j, sigid) as malformed, and that user finishes this function without outputting $(\text{failure}, j)$, then **Sim** halts.

This is indistinguishable from the sixth game. By the previous hybrid, either $\pi[k] = \text{sh}_{\text{index},j,k}$ for all $k \in B_{\text{msg}}$ or **Sim** has already halted. In the former case, if the honest player does not output failure then its check passed, so $\sigma_{\text{index},j}^L[k] = \pi[k] = \text{sh}_{\text{index},j,k}$ for every $k \in B_{\text{msg}}$. This is exactly what an honest party would output, so (j, sigid) would not have been marked as “malformed”.

Eighth Game: If no (j, sigid) is marked as “Malformed”, and there are at least T messages $(\text{TLOTS.Sign}_2, \text{sigid}, \sigma_{\text{index},j}^L)$ and **TLOTS.ReconstructSignature** $(\{\sigma_{\text{index},j}^L\})$ fails for these partial signatures or outputs an invalid signature, then **Sim** halts.

This never occurs by the properties of Shamir secret sharing. As noted above, if no partial signatures are malformed, then $\sigma_{\text{index},j}^L[k] = \text{sh}_{\text{index},j,k}$ for all $k \in B_{\text{msg}}$ (where j indicates which player provided the partial signature). By the fourth and seventh hybrids, all T partial signatures use the same **index**, so they are shares of a single secret key $\text{sk}_{\text{index}}^L$, where $\text{sk}_{\text{index}}^L = (\text{sh}_{\text{index},k})_{k=1}^e$ is the original secret key generated during key generation.

From here, correctness of CFF Lamport signatures is straightforward.

Ninth Game: Here **Sim** scans all messages sent by the adversary for values σ that could be valid signatures, i.e., it checks if σ is a valid signature for any message that has been added to the bulletin board. If it is a valid message, and fewer than $T - C$ users have output partial signatures for that message (where C is the number of currently corrupted users), **Sim** halts.

A valid signature must fit with a particular **LOTS** keypair; by the fourth hybrid, there is at most one other message msg' signed by this keypair.

We follow the approach of [5] to show that this breaks either **SM – TCR** or **SM – DSPR**. More precisely, we give two **SM – TCR** adversaries and a **SM – DSPR** adversary.

Our first **SM – TCR** adversary will run **Sim** for the 9th game, using **Adv**, and if **Sim** halts because of the valid signature above, it will check for a distance valid message msg' as mentioned above. It will output msg and msg' as its collision. It succeeds if $H_{\text{msg}}(\text{index}, \text{msg}) = H_{\text{msg}}(\text{index}, \text{msg}')$ (note that by the fourth hybrid H_{msg} is never called twice with the same tweak value **index**).

Our second **SM** – **TCR** adversary will do precisely the same, but it will instead halt if $H_{\text{msg}}(\text{index}, \text{msg}) = H_{\text{msg}}(\text{index}, \text{msg}')$. If it does not halt, then $B_{\text{msg}} = B_{\text{msg}'}$ and thus there is an index $k \in [e]$ such that the unexpected valid signature σ contains a preimage x of $\text{pk}^L[\text{index}][k]$. Hence this **SM** – **TCR** solver will output x and $\text{sk}^L[\text{index}][k]$ (as generated by **Sim**) as its collision.

Finally, we construct a **SM** – **DSPR** solver which also recovers the preimage x as mentioned above. If $x = \text{sk}_{\text{index},k}^L$, the **SM** – **DSPR** solver outputs $(0, \text{index}||k)$, and otherwise outputs $(1, \text{index}||k)$.

We see that because g_λ is injective, $B_{\text{msg}} = B_{\text{msg}'}$ if and only if the first **SM** – **TCR** solver succeeds. Hence, the probability of the spurious signature can be bounded above by $2\text{Adv}^{\text{SM-TCR}}(\mathcal{H}) + 2P$, where P is the probability of both a spurious signature on msg and for $B_{\text{msg}} \neq B_{\text{msg}'}$ (or no such msg' exists).

Suppose $B_{\text{msg}} \neq B_{\text{msg}'}$, or no such msg' exists. Then there exists at least one index $k \in [e]$ such that **Adv** has received fewer than $T - C$ shares of $\text{sk}^L[\text{index}][k]$. Then to be a valid signature, σ must contain a preimage x of $\text{pk}^L[\text{index}][k]$. If $x \neq \text{sk}^L[\text{index}][k]$, then the second **SM** – **TCR** solver succeeds (again, noting that the tweaks are all distinct for this function, since they are based on the index of the keypair and the index $k \in [e]$ into the pair).

We note here that *which* preimage of $\text{pk}^L[\text{index}][k]$ that **Sim** used is information-theoretically hidden from **Adv**, since Shamir secret sharing is information-theoretically hiding, and by definition **Adv** requested fewer than $T - C$ partial signatures for this message, and can produce at most C partial signatures itself, and only partial signatures provide shares of this preimage.

Thus, we can follow the same logic as [5, Claim 23]. We omit the full probability analysis and conclude that $P \leq 3\text{Adv}^{\text{SM-TCR}}(\mathcal{H}) + \text{Adv}^{\text{SM-DSPR}}$.

Tenth Game: This game uses a partial ideal functionality $\mathcal{F}_3$ which adds the Signing functionality to $\mathcal{F}_2$. This means that the honest parties will now forward **sign** requests sent from **UC.Env** directly to $\mathcal{F}_3$, instead of forwarding them to **Sim**.

To simulate this, as soon as any honest user j receives a **(sign, sigid, msg)** message, **Sim** will delay this message and run **TLOTS.Sign** locally. After this, as soon as **(sigid, msg)** is posted to the bulletin board, **Sim** will send **(sign, sigid, msg)** from all corrupt players. **Sim** will wait until j outputs a partial signature for **(sigid, msg)**, and then forward the original **sign** message.

When an honest user run by **Sim** outputs a **(signature, sigid, σ)** message, **Sim** will not write this to its output channel (i.e., what is visible to **UC.Env**) and instead send **(signature, sigid, msg, σ , $\beta = 1$, $\mathcal{C} = \emptyset$)** to $\mathcal{F}_3$. When $\mathcal{F}_3$ sends a **signature** message to an honest user j , **Sim** holds this message and does not deliver it until its internal user j outputs a **signature** message.

If an honest user j run by **Sim** outputs a **(failure, sigid, k)** message for some signature **sigid** on message **msg**, **Sim** will similarly not send this to the output channel, but compute the signature σ' that j computed internally, and send **(signature, sigid, msg, σ' , 0, k)** to $\mathcal{F}_3$. **Sim** will delay all **(failure, sigid, k)** messages for each honest party i until its internal execution of the real protocol results in i outputting **(failure, sigid, k)**.

We now argue that this is indistinguishable. First, we claim that the pairs $(\text{sigid}, \text{msg})$ recorded by $\mathcal{F}_3$ match precisely the list of $(\text{sigid}, \text{msg})$ given by the first messages on the bulletin board with a given sigid . To see this, we see that $\mathcal{F}_3$ stores $(\text{sigid}, \text{msg})$ only from the first user to send a given sigid, msg . Similarly, the bulletin board will store $(\text{sigid}, \text{msg})$ as the first message from sigid only from the first user to have their $(\text{sigid}, \text{msg})$ finalized. By construction **Sim** sends messages to $\mathcal{F}_3$ precisely when such messages are added to the bulletin board. Thus, these lists are the same.

UC.Env can see messages sent to honest players from $\mathcal{F}_3$, and **Sim** delays messages from $\mathcal{F}_3$ to match the real protocol; thus, the new ideal functionality cannot send *extra* messages that UC.Env would not have seen in the previous game when interacting with $\mathcal{F}_2$. The only difference could occur if an honest user sent an unexpected, unmatched message to UC.Env in the real protocol.

However, when an internally-simulated user j sends a $(\text{signature}, \text{sigid}, \sigma)$, **Sim** sends this to $\mathcal{F}_3$, who will then send a **signature** message back to that user (and all others), which **Sim** can immediately deliver. The only way this will not occur is if $\mathcal{F}_3$ halts because there is already a message recorded with that signature, or no record of that $(\text{sigid}, \text{msg})$ pair at all.

We first see by tracing execution of the real protocol that a user will only output a partial signature for the first message on the bulletin board with that sigid , and as previously argued this matches what $\mathcal{F}_3$ stores. We also see that users only produce finalized signature messages after they receive at least T partial signatures for that $(\text{sigid}, \text{msg})$ pair, and **Sim** sends a **sign** for each partial signature (notice that an honest user never reaches the execution of producing a partial signature without a **sign** request). Thus, there must be a record of $(\text{sigid}, \text{msg})$ must exist.

If the existing record has $\beta = 1$, then $\mathcal{F}_3$ would have already sent a **signature** message to user j and **Sim** can simply deliver it. If the existing record has $\beta = 0$ this would cause a problem. However, that only happens if **Sim** sent a message with that same signature and $\beta = 0$, which it will only do when a user outputs **failure**, which can only occur if σ is *not* a valid signature. Since all honest users run the same deterministic verification algorithm, and the bulletin board ensures that there is only one message associated to a given sigid , a given triple $(\text{sigid}, \text{msg}, \sigma)$ cannot be both valid and invalid. Thus, there is no problem here either.

When a user i outputs a $(\text{failure}, \text{sigid}, \sigma, j)$ message, **Sim** also sends this to $\mathcal{F}_3$. The same reasoning implies a $(\text{sigid}, \text{msg})$ record of some kind already exists, and the same reasoning implies either it contains no σ or it contains σ and $\beta = 0$. In either case, $\mathcal{F}_3$ will return a **failure** message to this user, and it will contain the same user ID j *unless* j was not a corrupted user. However, it is straightforward to see that the verification checks that honest users run will always pass for the values sent by other honest users, so the value j that $\mathcal{F}_3$ returns must match the value of j that the honest user originally sent.

Thus, this is distributed identically to the previous game.

Eleventh game: In the final game we use the actual ideal functionality $\mathcal{F}_{\text{Sig}}$. The only new component is verification. All that **Sim** must do to simulate this is to intervene if a verification message is sent to $\mathcal{F}$ with a pk that has not been recorded. If such a message occurs, **Sim** will delay it and send (**initialize**, **TLOTS.Verify**, pk) to $\mathcal{F}$, then send the verification message.

To prove that it is indistinguishable, we see that when $\mathcal{F}_{\text{Sig}}$ runs $\mathcal{V}$ – which **Sim** always supplies as **TLOTS.Verify** – this will match the real protocol perfectly. We are thus only concerned with the case where $\mathcal{F}_{\text{Sig}}$ receives a request where $\text{pk} = \text{pk}'$, the global state is uncompromised, there is no record for this (sigid, msg) pair, and $\mathcal{V}(\text{pk}, \sigma, \text{msg}) = 1$.

If there is no record for this (sigid, msg) pair, that means fewer than $T - C$ honest users have output a partial signature for this message, by simulation of **sign** messages. Thus, we can appeal to the previous hybrid argument, as we satisfy all the requirements: we have a valid signature and not enough partial signatures on it. Hence, this situation cannot occur anymore by the previous hybrid.

C.1 Bulletin Board

To prove security of the bulletin board, we first prove a few useful lemmas about consistency.

Lemma 1. *Let state_i be the internal state of an honest user i . Then for any k , if $\text{state}_i.B[k] \neq \perp$:*

1. $\text{state}_i.B[k][[h]] = H_{\text{BB}}(k - 1, \text{state}_i.B[k - 1])$
2. Every signature in $\text{state}_i.B[k][\text{sigs}]$ is a valid signature on $(\text{state}_i.B[k][[h]], \text{state}_i.B[k][[m]], \text{state}_i.B[k][[\text{sender}]])$ for a public key in state.pk
3. $\text{state}_i.B[k][\text{sigs}]$ has at least $\frac{N+F+1}{2}$ signatures

Proof. A user only adds to $\text{state}_i.B$ during **BB.AddBlocks** or **BB.SignedBlock**. During **BB.AddBlocks**, a block is added only after calling **BB.ValidateBlock**, which checks precisely the conditions above.

During **BB.SignedBlock**, the next block contains h computed to be $H_{\text{BB}}(k - 1, \text{state}_i.B[k - 1])$ during **BB.CreateAndShare**, and all the signatures are validated during execution, giving the result.

Corollary 1. *If two honest users call **BB.ValidateBlock** on the same input, they will return the same result.*

Proof. Since it is the same function, it can only differ based on the internal state, which depends only on the array state.pk , which is identical for all honest users thanks to $\mathcal{F}_{\text{DKG}}^{\text{BB}}$.

Lemma 2. *Let state_i and state_j be the states of two honest users i and j . If at most F parties are corrupt, and $\text{state}_i.B[k] \neq \perp$ and $\text{state}_j.B[k] \neq \perp$, then $\text{state}_i.B[k] = \text{state}_j.B[k]$.*

Proof. We notice that an honest user will never sign two different blocks for a particular index k . This can be seen by inspecting **BB.SignInternal**: after a user signs a block at index μ , they add this to `state.signed` $[\mu]$, and before signing, users check that `state.signed` $[\mu]$ does not contain a different block.

By Lemma 1, an honest user will only add an element to `state.B` if it has at least $\frac{N+F+1}{2}$ signatures. This implies the number of honest users who have added signatures is at least $\frac{N+F+1}{2} - C \geq \frac{N+F-2C+1}{2}$, where C is the number of corrupt users.

Combining these two conclusions, if a user i and j have distinct blocks at the same index, since no honest user signed both blocks, the number of honest signers in total is at least $N + F - 2C + 1$. Since $F \geq C$, this is at least $N - C + 1$ – but there are only $N - C$ honest users, a contradiction.

This leads to an easy corollary:

Corollary 2. *If two honest users return a non- $\perp$ value from **BB.Read**, they return the same value.*

Lemma 3. ***BB.Read**(i) will only ever return values `msg` such that `state.B` $[i][[m]] = \text{msg}$ for all honest users with `state.B` $[i] \neq \perp$.*

Proof. By Lemma 2, all honest users agree on `state.B` $[i]$, so we will refer to this value unambiguously. We can see that if an honest user runs **BB.Read**(i) and has a value for `state.B` $[i]$ it will return it immediately. Otherwise, the caller will wait for at least $\frac{N+F+1}{2}$ responses. Suppose after M responses there is a majority response. For this to not equal `state.B` $[i]$, all M responses must be corrupt, meaning $\frac{N+F+1}{4} < M \leq C$, or $N \leq 4C - F - 1 \leq 3F - 1 \leq N - 1$ (since $F \leq N/3$), a clear contradiction.

If there are N responses, there must be a majority response, since there are at least $N - F \geq \frac{2}{3}N$ honest parties.

Theorem 4. *Our bulletin board UC-securely realizes $\mathcal{F}_{\text{BB}}$.*

Proof. Our simulator works as follows: it will internally run **Adv** and all honest users.

When **Sim** reads any message sent to a dummy honest user j (which forwards it immediately to $\mathcal{F}_{\text{BB}}$), **Sim** will record the message and the ID of the user. When it receives the response from $\mathcal{F}_{\text{BB}}$, **Sim** will send the corresponding function message to its internal honest user j .

Sim will track all honest users' internal variables `state.B`. Once a single honest user adds an i th block to `state.B`, **Sim** will send (`state.B` $[i][[m]]$) to $\mathcal{F}_{\text{BB}}$. This is unambiguous because all honest users agree on `state.B` $[i]$ thanks to Lemma 2.

When **Sim** receives a (**BB.Read**, i) message sent to its internal router, it will simulate the corresponding internal **BB.Read** of the real protocol until this returns a value, at which point **Sim** will send the original message to $\mathcal{F}_{\text{BB}}$ and immediately send its response to the original requesting party.

Sim will perform the same type of simulation for **BB.Search**.

If the global state becomes compromised, then on any **BB.Read** or **BB.Search** request, **Sim** will simply follow the real protocol and return whatever answer this delivers to $\mathcal{F}_{\text{BB}}$.

Now we argue that this is an indistinguishable simulation. By Lemma 3, **BB.Read** only ever returns a value equal to the block in `state.B` for honest parties, which matches what is added to the ideal functionality. We see that in the real execution, **BB.Read**(i) does not return until at least one honest party has added an i th block, but when this happens $\mathcal{F}_{\text{BB}}$ has a value for $B[i]$ as well. After that point, **Sim** delays messages to precisely match the message delivery times of the real protocol.

BB.Search is effectively a helper function for **BB.Read**, so its behaviour is the same.