Practical Cryptography for Blockchains: Minimizing Trust and Maximizing Security

Noemi Glaeser

Abstract

In 2008, Satoshi Nakamoto introduced Bitcoin, the first digital currency without a trusted authority whose security is maintained by a decentralized blockchain. Since then, a plethora of decentralized applications have been proposed utilizing blockchains as a public bulletin board. In recent years, it has become clear that this basic functionality is not enough to prevent widespread attacks on both the privacy and security of blockchain users, as evidenced by the blockchain analytics industry and the billions of dollars stolen via cryptocurrency exploits to date.

This work explores the role cryptography has to play in the blockchain ecosystem to both reduce trust and secure user funds. I discuss a new paradigm for rollups called "naysayer proofs" which sits between optimistic and zero-knowledge approaches. Next, I show how to construct efficient, non-interactive, and private on-chain protocols for a large class of elections and auctions. Finally, I introduce a new type of threshold wallet which adds offline components at each custodian to improve security. Furthermore, the construction enables transparency by allowing the wallet owner to request custodians to prove "remembrance" of the secret key material. All three of these works will be deployed in practice or have seen interest from practitioners who I hope will be interested in applying them. Thus, they have real practical impact. I conclude by describing future directions.

Acknowledgments

Basis of this Dissertation

[GGJ$^+$24] Sanjam Garg, Noemi Glaeser, Abhishek Jain, Michael Lodder, and Hart Montgomery. Hot-cold threshold wallets with proofs of remembrance, 2024. *[page 20.]*

[GSZB23] Noemi Glaeser, István András Seres, Michael Zhu, and Joseph Bonneau. Cicada: A framework for private non-interactive on-chain auctions and voting. Cryptology ePrint Archive, Paper 2023/1473, 2023. https://eprint.iacr.org/2023/1473.

[SGB24] István András Seres, Noemi Glaeser, and Joseph Bonneau. Short paper: Naysayer proofs. In Jeremy Clark and Elaine Shi, editors, *Financial Cryptography and Data Security*, 2024. *[page 12.]*

Other Publications by the Author

[AGRS24] Behzad Abdolmaleki, Noemi Glaeser, Sebastian Ramacher, and Daniel Slamanig. Circuit-succinct universally-composable NIZKs with updatable CRS. In *37th IEEE Computer Security Foundations Symposium*, 2024.

[GKMR23] Noemi Glaeser, Dimitris Kolonelos, Giulio Malavolta, and Ahmadreza Rahimi. Efficient registration-based encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 1065–1079. ACM Press, November 2023.

[GMM$^+$22] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri Aravinda Krishnan Thyagarajan. Foundations of coin mixing services. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1259–1273. ACM Press, November 2022.

Contents

1 Introduction

Bitcoin [Nak08] was the first digital currency to successfully implement a fully trustless and decentralized payment system. Underpinning Bitcoin is a *blockchain*, a distributed append-only ledger to record transactions. In Bitcoin, the blockchain's consistency is enforced via a "proof-of-work" (PoW) consensus mechanism in which participants solve difficult computational puzzles (hash preimages) to append the newest bundle of transactions (a block) to the chain.

Ethereum [But14] introduced programmability via *smart contracts*, special applications which sit on top of the consensus layer and can maintain state and modify it programmatically. This has led to the emergence of a number of *decentralized applications* enabling more diverse functionalities in a trustless manner.

2 Preliminaries

We use $[n]$ to denote the range $\{1, \ldots, n\}$. For other ranges (mostly zero-indexed), we explicitly write the (inclusive) endpoints, e.g., $[0, n]$. Concatenation of vectors $\mathbf{x}, \mathbf{y}$ is written as $\mathbf{x}||\mathbf{y}$. Let λ be the security parameter. We use the uppercase variable X for the free variable of a polynomial, e.g., $f(X)$. We use a calligraphic font, e.g., $\mathcal{S}$ or $\mathcal{X}$, to denote sets or domains. When we apply an operation to two sets of equal size ℓ we mean pairwise application, e.g., $\mathcal{Z} = \mathcal{X} + \mathcal{Y}$ means $z_i = x_i + y_i\ \forall i \in [\ell]$. Sampling an element x uniformly at random from a set $\mathcal{X}$ is written as $x \overset{\$}{\leftarrow} \mathcal{X}$. We use $:=$ to denote variable assignment, $y \leftarrow \mathsf{Alg}(x)$ to assign to y the output of some algorithm Alg on input x, and $y \overset{\$}{\leftarrow} \mathsf{Alg}(x)$ if the algorithm is randomized. When we wish to be explicit about the randomness r used, we write $y \leftarrow \mathsf{Alg}(x; r)$.

For a non-interactive zero-knowledge proof system Π, we write $\pi \leftarrow \Pi.\mathsf{Prove}(x; w)$ to show that the proving algorithm takes as input an instance x and witness w and outputs a proof π. Verification is written as $\Pi.\mathsf{Vrfy}(x, \pi)$ and outputs a bit b.

We distinguish the key-pairs used in a signature scheme (vk, sk for "verification" and "signing" key, respectively) from those used in an encryption scheme (ek, dk for "encryption" and "decryption" key, respectively).

2.1 Non-interactive proof systems

Definition 1 (Non-interactive proof)**.** *A non-interactive proof* Π *for some NP relation* $\mathcal{R}$ *is a tuple of PPT algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$*:*

$\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{crs}$**:** *Given a security parameter, output a common reference string* crs. *This algorithm might use private randomness (a trusted setup).*

$\mathsf{Prove}(\mathsf{crs}, x, w) \rightarrow \pi$**:** *Given the* crs*, an instance* x*, and witness* w *such that* $(x, w) \in \mathcal{R}$*, output a proof* π*.*

$\mathsf{Vrfy}(\mathsf{crs}, x, \pi) \to \{0, 1\}$**:** *Given the* crs *and a proof* π *for the instance* x*, output a bit indicating accept or reject.*

We refer the reader to [Tha23] for a formal description of the properties of proof systems (e.g., correctness, soundness, zero-knowledge).

2.2 Bilinear Pairings

Definition 2 (bilinear pairing)**.** *A bilinear pairing is a map* $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ *where* $\mathbb{G}_1, \mathbb{G}_2$*, and* $\mathbb{G}_T$ *are cyclic groups of prime order* p*. (Often,* $\mathbb{G}_1, \mathbb{G}_2$ *are written in additive notation and* $\mathbb{G}_T$ *in multiplicative notation.) Let* $g_1, h_1 \in \mathbb{G}_1$ *and* $g_2, h_2 \in \mathbb{G}_2$ *(generators of their respective groups). The map* e *has the following properties:*

Bilinearity: *For all* $a, b \in \mathbb{Z}_p^*$*, the following hold:*

$$\begin{aligned} e(g_1^a, g_2) = e(g_1, g_2^a) &= e(g_1, g_2)^a \\ e(g_1 h_1, g_2) &= e(g_1, g_2) \cdot e(h_1, g_2) \\ e(g_1, g_2 h_2) &= e(g_1, g_2) \cdot e(g_1, h_2) \end{aligned}$$

Non-degeneracy: $e(g_1, g_2) \neq 1$*.*

Computability: *There is an efficient algorithm for computing* e*.*

Pairings are divided into types based on the (in)equality of the utilized groups $\mathbb{G}_1, \mathbb{G}_2$ and whether there is an efficiently computable homomorphism between the groups. For our purposes, we will use a type-3 pairing: asymmetric ($\mathbb{G}_1 \neq \mathbb{G}_2$) and with no such efficiently computable homomorphism. In this case, the Decisional Diffie-Hellman (DDH) assumption is believed to hold in both $\mathbb{G}_1$ and $\mathbb{G}_2$; this is referred to as the *symmetric external Diffie-Hellman (SXDH) assumption.*

2.3 Shamir Secret Sharing

Shamir [Sha79] introduced a scheme to share a secret among n parties such that any t parties can work together to recover the secret, but with any fewer parties the secret remains information-theoretically hidden.

Construction 1 (Shamir secret sharing [Sha79])**.** *Let* p *be a prime.*

- $\underline{\{s_1, \ldots, s_n\} \leftarrow \mathsf{Share}(s, t, n)}$*: Given a secret* $s \in \mathbb{Z}_p$ *and* $t \leq n \in \mathbb{N}$*, compute a* t*-out-of-*n *sharing of* s *by choosing a random degree-*$(t-1)$ *polynomial* $f(X) \in \mathbb{Z}_p[X]$ *such that* $f(0) = s$*. For* $i \in [n]$*, compute* $s_i := (i, f(i))$*.*
- $\underline{\{s', \bot\} \leftarrow \mathsf{Reconstruct}(S, t, n)}$*: Given some set of shares* S*, check if* $|S| < t$*. If so, output* $\bot$*. Otherwise, without loss of generality, let* $S' := \{s_1, \ldots, s_t\}$ *be the first* t *entries of* S*, where* $s_i := (x_i, y_i)$*. Output the Lagrange interpolation at 0:*

$$s' := \sum_{i=1}^{t} y_i \prod_{j=1, j\neq i}^{t} \frac{x_j}{x_j - x_i}.$$

The secret sharing scheme is *correct*, since for any secret s and values $t \leq n \in \mathbb{N}$, we have $\mathsf{Reconstruct}(\mathsf{Share}(s,t,n),t,n) = s$.

For notational convenience, let $\mathsf{Share}(s,t,n;r)[i]$ denote the ith share of s computed with randomness r. The reconstruction algorithm can be generalized to interpolate any point $f(k)$ (not just the secret at $k = 0$) and thereby recover the ith share:

- $\underline{\{s_k, \perp\} \leftarrow \mathsf{Interpolate}(S,k,t,n)}$: If $|S| < t$, output $\perp$. Otherwise, use the first t entries $(x_1,y_1),\ldots,(x_t,y_t)$ of S to interpolate

$$f(k) = \sum_{i=1}^{t} y_i \prod_{j=1, j\neq i}^{t} \frac{x_j - k}{x_j - x_i}.$$

  Output $s_k := (k, f(k))$.

2.4 BLS Signatures

Construction 2 (BLS signature scheme [BLS01])**.** *Let $\mathbb{G}_1, \mathbb{G}_2$ be elliptic curve groups of order p generated by g_1 and g_2, respectively, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable asymmetric pairing between them. We also require a hash function $H : \{0,1\}^* \to \mathbb{G}_1$. The signature scheme works as follows:*

- $\underline{(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda)}$*: Given the security parameter 1^λ, sample $x \xleftarrow{\$} \mathbb{Z}_p$ and output the keypair consisting of signing key $\mathsf{sk} := x$ and verification key $\mathsf{vk} := g_2^x$.*

- $\underline{\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)}$*: Given a signing key $\mathsf{sk} \in \mathbb{Z}_p$ and a message $m \in \{0,1\}^*$, compute a signature $\sigma := H(m)^{sk}$.*

- $\underline{\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{vk}, m, \sigma)}$*: Given a verification key $\mathsf{vk} \in \mathbb{G}_2$, message $m \in \{0,1\}^*$, and signature $\sigma \in \mathbb{G}_1$, if $e(\sigma, g_2) = e(H(m), \mathsf{vk})$, output 1. Else output 0.*

The security of BLS relies on the gap co-Diffie Hellman assumption on $(\mathbb{G}_1, \mathbb{G}_2)$, i.e., co-DDH being easy but co-CDH being hard on $\mathbb{G}_1, \mathbb{G}_2$, as well as the existence of an efficiently computable homomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$ (type-2 pairing). Since we require a type-3 pairing for our purposes (i.e., no efficiently computable ϕ exists), we rely on a stronger variant of the co-GDH assumption (see discussion in [BLS01, §3.1] and [SV05, §2.2]).

Threshold variant Sharing a BLS signing key $\mathsf{sk} \in \mathbb{Z}_p$ via Shamir secret sharing leads directly to a robust t-out-of-n threshold signature [BLS01]. More specifically, each party $i \in [n]$ receives a t-out-of-n Shamir secret share sk_i of the key. The "partial" public keys $\mathsf{vk}_i := g_2^{\mathsf{sk}_i}$ are published along with the public key vk.

A partial signature is computed in exactly the same way as a regular BLS signature, but under the secret key share: $\sigma_i := H(m)^{\mathsf{sk}_i}$. This value is publicly verifiable by checking that $(g_2, \mathsf{vk}_i, H(m), \sigma_i)$ is a co-Diffie-Hellman tuple (i.e., it is of the form (g_2, g_2^a, h, h^a) where $g_2 \in \mathbb{G}_2$ and $h \in \mathbb{G}_1$).

Given t valid partial signatures on a message $m \in \{0,1\}^*$ anyone can recover a regular BLS signature:

- $\underline{\sigma \leftarrow \mathsf{Reconstruct}(S := \{(i, \sigma_i)\})}$: Let $S' \subseteq S$ be the set of valid partial signatures in S. If $|S'| < t$, output $\bot$. Otherwise, without loss of generality, assume the first t valid signatures come from users $1, \dots, t$ and recover the complete signature as
$$\sigma \leftarrow \prod_{i=1}^{t} \sigma_i^{\lambda_i}, \text{ where } \lambda_i = \prod_{j=1, j\neq i}^{t} \frac{j}{j-i} \pmod p$$

Notice that the reconstruction simply performs Shamir reconstruction of the signing key shares sk_i in the exponent and thus the output will equal $H(m)^{\mathsf{sk}}$. Hence, the complete signature is indistinguishable from a regular BLS signature, and verification proceeds exactly as in the regular scheme.

2.5 KZG polynomial commitments

We give the asymmetric version of KZG below.

Construction 3 (KZG polynomial commitments [KZG10])**.** *Let $\mathbb{G}_1, \mathbb{G}_2$ be elliptic curve groups of order p with generators g_1, g_2 and $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ be an elliptic curve pairing. The following is a commitment scheme for polynomials in $\mathbb{Z}_p[X]$ with degree at most d.*

- $\underline{\mathsf{crs} \leftarrow \mathsf{Setup}(d):}$ *Sample* $\tau \xleftarrow{\$} \mathbb{Z}_p$ *and output* $\mathsf{crs} := \{g_1, g_1^{\tau}, g_1^{\tau^2}, \dots, g_1^{\tau^d}, g_2, g_2^{\tau}\}$.

- $\underline{\mathsf{com}_f \leftarrow \mathsf{Com}(\mathsf{crs}, f(X)):}$ *Let* $f(X) = a_0 + a_1 X + \cdots + a_d X^d \in \mathbb{Z}_p[X]$*. Use* crs *to compute and output* $g_1^{f(\tau)} = g_1^{a_0} \cdot (g_1^{\tau})^{a_1} \dots (g_1^{\tau^d})^{a_d} = g_1^{a_0 + a_1\tau + \cdots + a_d \tau^d} \in \mathbb{G}_1$.

- $\underline{(f(i), \pi_i) \leftarrow \mathsf{Open}(\mathsf{crs}, f(X), i):}$ *To open* $f(X)$ *at* i*, let* $q_i(X) := \frac{f(X) - f(i)}{X - i} \in \mathbb{Z}_p[X]$[1]*. Then compute* $\mathsf{com}_{q_i} \leftarrow \mathsf{Com}(\mathsf{crs}, q_i(X))$ *and output* $(f(i), \mathsf{com}_{q_i}) \in \mathbb{Z}_p \times \mathbb{G}_1$.

- $\underline{\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{crs}, \mathsf{com}_f, i, y, \pi_i):}$ *To confirm* $y = f(i)$*, it suffices to check that* $q_i(X) = \frac{f(X) - y}{X - i}$ *at* $X = \tau$*. This can be done with a single pairing check:*
$$e(\mathsf{com}_f / g_1^y, g_2) \stackrel{?}{=} e(\pi_i, g_2^{\tau} / g_2^{i})$$

The security of the scheme relies on the d-Strong Diffie Hellman assumption (d-SDH), which states that given $\{g_1, g_1^{\tau}, \dots, g_1^{\tau^d}, g_2, g_2^{\tau}\}$, it is difficult to compute $(c, g_1^{\frac{1}{\tau - c}})$ for any $c \in \mathbb{Z}_p \setminus \{-\tau\}$. This assumption is stronger than d-SDH in $\mathbb{G}_1$, which in turn implies DDH in $\mathbb{G}_1$.

[1]This is a polynomial by Little Bézout's Theorem.

2.6 Pedersen Commitments

Next, we recall Pedersen commitments [Ped92], a commitment scheme which is unconditionally (information-theroetically) hiding and computationally binding (by the discrete logarithm assumption on $\mathbb{G}$). In our construction we will instantiate the scheme over $\mathbb{G}_1$, so we let $\mathbb{G} = \mathbb{G}_1$ in the description below.

Construction 4 (Pedersen commitment scheme [Ped92])**.** *Let $\mathbb{G}_1$ be a group of order p and g_1, h_1 be generators of $\mathbb{G}_1$. The following is a commitment scheme for elements $x \in \mathbb{Z}_p$.*

- $\underline{(\mathsf{com}, \mathsf{decom}) \leftarrow \mathsf{Com}(x) :}$ *Sample $r \overset{\$}{\leftarrow} \mathbb{Z}_p$ and return $\mathsf{com} := g_1^x h_1^r$ and decommitment information (x, r).*
- $\underline{(x, r) \leftarrow \mathsf{Open}(\mathsf{com}, \mathsf{decom}) :}$ *To open com, directly output $\mathsf{decom} = (x, r)$.*
- $\underline{\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{com}, x, r) :}$ *To confirm the opening of com to x, it suffices to check that $\mathsf{com} = g_1^x h_1^r$.*

A PoK of the committed value can be computed using a Sigma protocol due to Okamoto [Oka93], which can be made non-interactive using the Fiat-Shamir transform [FS87]. We refer to this protocol as Π_{ped} and present it in Figure 1.

PoK of Pedersen opening (Π_{ped})

Parameters: Group $\mathbb{G}_1$ of order p with generators g_1, h_1.

$\underline{\mathsf{Prove}(\mathsf{com}_{\mathsf{ped}}; (v, r)) \to \pi_{\mathsf{ped}}}$: Given a Pedersen commitment $\mathsf{com}_{\mathsf{ped}} = g_1^v h_1^r$, a party can prove knowledge of the opening (v, r) as follows:

1. The prover samples $s_1, s_2 \overset{\$}{\leftarrow} \mathbb{Z}_p$ and sends $a := g_1^{s_1} h_1^{s_2}$ to the verifier.
2. The verifier sends back a uniform challenge $c \overset{\$}{\leftarrow} \mathbb{Z}_p$.
3. The prover computes $t_1 := s_1 + vc$ and $t_2 := s_2 + rc$ and sends both values to the verifier.

The proof is defined as $\pi_{\mathsf{ped}} := (a, c, (t_1, t_2))$.

$\underline{\mathsf{Vrfy}(\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}) \to \{0,1\}}$: Given the commitment $\mathsf{com}_{\mathsf{ped}}$ and a proof π_{ped}, the verifier parses $\pi_{\mathsf{ped}} := (a, c, (t_1, t_2))$ and outputs 1 iff $a \cdot \mathsf{com}_{\mathsf{ped}}^c = g_1^{t_1} h_1^{t_2}$.

Figure 1: The proof system Π_{ped} used to prove knowledge of the opening to a Pedersen commitment [Oka93].

2.7 Leftover Hash Lemma

We use the presentation of the leftover hash lemma (LHL) [IZ89] from [AMPR19].[2] Let $(\mathcal{X}, \oplus)$ be a finite group of size $|\mathcal{X}|$, and let n be a positive integer. For any fixed $2n$-vector of group elements $\mathbf{x} = \{x_{j,b}\}_{j\in[n],b\in\{0,1\}} \in \mathcal{X}^{2n}$, denote by $\mathcal{S}_{\mathbf{x}}$ the following distribution:

$$\mathcal{S}_{\mathbf{x}} = \left\{ \bigoplus_{j\in[n]} x_{j,r_j} : (r_1, \cdots, r_n) \leftarrow \{0,1\}^n \right\}.$$

Also, let $\mathcal{U}_{\mathcal{X}}$ denote the uniform distribution over $\mathcal{X}$, and let $\Delta(\mathcal{D}_1, \mathcal{D}_2)$ denote the statistical distance between the distributions $\mathcal{D}_1$ and $\mathcal{D}_2$. We will use the following special case of leftover hash lemma [IZ89]. The proof can be found in the JoC version of [AMPR19].

Lemma 1. (Leftover Hash Lemma.) *Let $(\mathcal{X}, \oplus)$ be a finite group, and let $\mathcal{S}_{\mathbf{x}}$ and $\mathcal{U}_{\mathcal{X}}$ be two distributions over $\mathcal{X}$ as defined above. For any (large enough) positive integer n, it holds that*

$$\Pr_{\mathbf{x}\leftarrow\mathcal{X}^{2n}}\left[\Delta(\mathcal{S}_{\mathbf{x}}, \mathcal{U}_{\mathcal{X}}) > \sqrt[4]{\frac{|\mathcal{X}|}{2^n}}\right] \leq \sqrt[4]{\frac{|\mathcal{X}|}{2^n}}.$$

In particular, for any $n > \log(|\mathcal{X}|) + \omega(\log(\lambda))$, if $\mathbf{x}$ is sampled uniformly then with overwhelming probability the statistical distance between two distributions is negligible.

2.8 Universal Composability (UC) Framework

In the universal composability (UC) framework [Can01], the security requirements of a protocol are defined via an *ideal functionality* which is executed by a trusted party. To prove that a protocol *UC-realizes* a given ideal functionality, we show that the execution of this protocol (in the real or hybrid world) can be *emulated* in the ideal world, where in both worlds there is an additional adversary $\mathcal{E}$ (the environment) which models arbitrary concurrent protocol executions. Specifically, we show that for any adversary $\mathcal{A}$ attacking the protocol execution in the real world (by controlling communication channels and corrupting parties involved in the protocol execution), there exists an adversary $\mathcal{S}$ (the simulator) in the ideal world who can produce a protocol execution which no environment $\mathcal{E}$ can distinguish from the real-world execution.

Below we describe the UC framework as it is presented in [CLOS02]. All parties are represented as probabilistic interactive Turing machines (ITMs) with input, output, and ingoing/outgoing communication tapes. For simplicity, we assume that all communication is authenticated, so an adversary can only delay but not forge or modify messages from parties involved in the protocol. Therefore, the order of message delivery is also not guaranteed (asynchronous

[2]We specifically use the improved version from the Journal of Cryptology version of this paper.

communication). We consider a PPT malicious, adaptive adversary who can corrupt or tamper with parties at any point during the protocol execution.

The execution in both worlds consists of a series of sequential party activations. Only one party can be activated at a time (by writing a message on its input tape). In the real world, the execution of a protocol Π occurs among parties $P_1, \ldots, P_n$ with adversary $\mathcal{A}$ and environment $\mathcal{E}$. In the ideal world, interaction takes place between dummy parties $\tilde{P}_1, \ldots, \tilde{P}_n$ communicating with the ideal functionality $\mathcal{F}$, with the adversary (simulator) $\mathcal{S}$ and environment $\mathcal{E}$. Every copy of $\mathcal{F}$ is identified by a unique session identifier `sid`.

In both the real and ideal worlds, the environment is activated first and activates either the adversary ($\mathcal{A}$ resp. $\mathcal{S}$) or an uncorrupted (dummy) party by writing on its input tape. If $\mathcal{A}$ (resp. $\mathcal{S}$) is activated, it can take an action or return control to $\mathcal{E}$. After a (dummy) party (or $\mathcal{F}$) is activated, control returns to $\mathcal{E}$. The protocol execution ends when $\mathcal{E}$ completes an activation without writing on the input tape of another party.

We denote with $\textsc{real}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)$ the random variable describing the output of the real-world execution of Π with security parameter λ and input x in the presence of adversary $\mathcal{A}$ and environment $\mathcal{E}$. We write the corresponding distribution ensemble as $\{\textsc{real}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda \in \mathbb{N}, x \in \{0,1\}^*}$. The output of the ideal-world interaction with ideal functionality $\mathcal{F}$, adversary (simulator) $\mathcal{S}$, and environment $\mathcal{E}$ is represented by the random variable $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)$ and corresponding distribution ensemble $\{\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda \in \mathbb{N}, x \in \{0,1\}^*}$.

The actions each party can take are summarized below:

- Environment $\mathcal{E}$: **read** output tapes of the adversary ($\mathcal{A}$ or $\mathcal{S}$) and any uncorrupted (dummy) parties; then **write** on the input tape of one party (the adversary $\mathcal{A}$ or $\mathcal{S}$ or any uncorrupted (dummy) parties).

- Adversary $\mathcal{A}$: **read** its own tapes and the outgoing communication tapes of all parties; then **deliver** a pending message to party by writing it on the recipient's ingoing communication tape *or* **corrupt** a party (which becomes inactive: its tapes are given to $\mathcal{A}$ and $\mathcal{A}$ controls its actions from this point on, and $\mathcal{E}$ is notified of the corruption).

- Real-world party P_i: only follows its code (potentially writing to its output tape or sending messages via its outgoing communication tape).

- Dummy party $\tilde{P}_i$: acts only as a simple relay with the ideal functionality $\mathcal{F}$, copying inputs from its input tape to its outgoing communication tape (to $\mathcal{F}$) and any messages received on its ingoing communication tape (from $\mathcal{F}$) to its output tape.

- Adversary $\mathcal{S}$: **read** its own input tape and the public headers (see below) of the messages on $\mathcal{F}$'s and dummy parties' outgoing communication tapes; then **deliver** a message to $\mathcal{F}$ from a dummy party or vice versa by copying it from the sender's outgoing communication tape to the recipient's incoming communication tape *or* **send** its own message to $\mathcal{F}$ by writing on the latter's incoming communication tape *or* **corrupt** a dummy party (which becomes inactive: its tapes are given to $\mathcal{S}$ and $\mathcal{S}$ controls its actions from this point on, and $\mathcal{E}$ and $\mathcal{F}$ are notified of the corruption).

- Ideal functionality $\mathcal{F}$: **read** incoming communication tape; then **send** any messages specified by its definition to the dummy parties and/or adversary $\mathcal{S}$ by writing to its outgoing communication tape.

Definition 3. *We say a protocol* Π UC-realizes *an ideal functionality* $\mathcal{F}$ *if for any PPT adversary* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that for any environment* $\mathcal{E}$*, the distribution ensembles* $\{\text{REAL}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$ *and* $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$ *are computationally indistinguishable.*

3 Naysayer Proofs

(Parts of this section are taken/adapted from [SGB24].)

In most blockchains with programming capabilities, e.g., Ethereum [Woo24], developers are incentivized to minimize the storage and computation complexity of on-chain programs. Applications with high compute or storage incur significant fees, commonly referred to as *gas*, to compensate validators in the network. Often, these costs are passed on to users of an application.

High gas costs have motivated many applications to utilize *verifiable computation* [GGP10], off-loading expensive operations to powerful but untrusted off-chain entities who perform arbitrary computation and provide a succinct non-interactive proof (SNARK) that the claimed result is correct. This computation can even depend on secret inputs not known to the verifier in the case of zero-knowledge proofs (i.e., zkSNARKs).

Verifiable computation leads to a paradigm in which smart contracts, while capable of arbitrary computation, primarily act as verifiers and outsource all significant computation off-chain. A motivating application is *rollups*, which combines transactions from many users into a single smart contract which verifies a proof that all have been executed correctly. However, verifying these proofs can still be costly. For example, the StarkEx rollup has spent hundreds of thousands of dollars to date to verify FRI polynomial commitment opening proofs.[3]

We observe that this proof verification is often wasteful. In most applications, provers have strong incentives to only post correct proofs, suffering direct financial penalties (in the form of a lost security deposit) or indirect costs to their reputation and business for posting incorrect proofs. As a result, a significant fraction of a typical layer-1 blockchain's storage and computation is expended verifying proofs, which are almost always correct.[4]

This state of affairs motivates us to propose a new paradigm called *naysayer proofs*. In this paradigm, the verifier (e.g., a rollup smart contract) optimisti-

[3] https://etherscan.io/address/0x3e6118da317f7a433031f03bb71ab870d87dd2dd

[4] At the time of this writing, we are unaware of any major rollup service which has posted an incorrect proof in production.

	VC	fraud proof (interactive)	fraud proof (non-interactive)	naysayer proof
No optimistic assumption	●	○	○	○
Non-interactive	●	○	●	●
Off-chain original Vrfy	○	●	●	●
Witness-independent resolution	●	◐	○	●

Table 1: Trade-offs of verifiable computation (VC), interactive (e.g., Arbitrum [Lab23], Optimism V2 [Opt23]) and non-interactive (e.g., Optimism V1) fraud proofs, and naysayer proofs.

cally accepts a submitted proof without verifying its correctness. Instead, any observer can check the proof off-chain and, if needed, prove its *incorrectness* to the verifier by submitting a *naysayer proof.* The verifier then checks the naysayer proof and, if it is correct, rejects the original proof. Otherwise, if no party can successfully naysay the original proof before the end of the dispute period, the original proof is accepted. To deter denial of service, naysayers may be required to post collateral, which is forfeited if their naysayer proof is incorrect.

This paradigm potentially saves the verifier work in two ways. First, in the optimistic case, where the proof is not challenged, the verifier does no work at all. We expect this to almost always be the case in practice. Second, even in the pessimistic case, checking the naysayer proof may be much more efficient than checking the original proof (e.g., if the verification fails at a single point, the naysayer proof can just point to that specific step). In other words, the naysayer acts as a helper to the verifier by reducing the cost of the verification procedure in fraudulent cases. At worst, checking the naysayer proof is equivalent to verifying the original proof (this is the trivial naysayer construction).

Naysayer proofs enable other interesting trade-offs. For instance, naysayer proofs might be run at a lower security level than the original proof system. A violation of the naysayer proof system's soundness undermines the *completeness* of the original proof system. For an application like a rollup service, this results only in a loss of liveness; importantly, the rollup users' funds would remain secure. Liveness could be restored by falling back to full proof verification.

In Section 3.2, we formally define naysayer proofs and show that every NP language has a logarithmic size and constant-time naysayer proof. In Section 3.3, we provide three concrete examples where naysayer proofs offer significant speedups.

3.1 Related Work

A concept related to the naysayer paradigm is *refereed delegation* [FK97]. The idea has found widespread adoption in the form of "fraud proofs" or "fault proofs" as used in emphoptimistic rollups [Eth23b, Lab23, Opt23, TR19]. Like naysayer proofs, fraud proofs work under an optimistic assumption, i.e., a computation is assumed to be correct unless some party challenges it. In case of a challenge, a dispute resolution process ensues between the *challenger* and the *defender*, which can be either non-interactive or interactive. In the former approach, the full computation is re-executed by the on-chain verifier to resolve the dispute. In the latter approach, the challenger and defender engage in a *bisection protocol* to locate a disputed step of the computation, and only that step is re-executed to resolve the dispute.

We compare classic verifiable computation, fraud proofs, and our new approach in Table 1. At a high level, in the fraud proof paradigm, a "prover" performs a provisionally accepted computation without any proof of correctness. Any party can then challenge the correctness of the prover's *computation*. In the naysayer paradigm, by contrast, the prover supplies a proof with the computation output, which is provisionally accepted. Any party can then challenge the correctness of the *proof*. The naysayer approach offers significant speedups since the verifier's circuit is typically much smaller than the original computation. Note that there is a slight semantic difference: fraud proofs can definitively show that the computation output is incorrect. In contrast, naysayer proofs can only show that the accompanying proof is invalid—the computation itself may have been correct.

Furthermore, for fraud proofs, the full computation input (the witness) must be made available to the verifier and potential challengers. Naysayer proofs, on the other hand, can be verified using only the statement and proof. Hence, naysayer proofs work naturally with zero-knowledge proofs. This can also lead to crucial savings if the witness is very large (e.g., transaction data for a rollup).

The fraud proof design pattern has been applied in an application-specific way in many blockchain applications besides optimistic rollups, including the Lightning Network [PD16], Plasma [PB17], cryptocurrency mixers [SNBB19], and distributed key generation [SJSW19]. We view naysayer proofs as a drop-in replacement for the many application-specific fault proofs, offering an alternative which is both more general and more efficient. There are three entities in a naysayer proof system. We assume that all parties can read and write to a public bulletin board (e.g. a blockchain).

Prover The prover posts a proof π to the bulletin board claiming $(x, w) \in \mathcal{R}$.

Verifier The verifier does not directly verify the validity of π, rather, it allows everyone to naysay in a pre-defined time window of duration T_{nay}. Optimistically, if no one naysays π within time T_{nay}, the verifier accepts it. In the pessimistic case, a party (or multiple parties) naysay the validity of π by posting proof(s) π_{nay}. The verifier checks the validity of each π_{nay}, and if any of them pass, it rejects the original proof π.

Naysayer If $\mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 0$, then the naysayer posts a naysayer proof π_{nay} to the public bulletin board before T_{nay} time elapses.

Note that we need to assume a synchronous communication model as we cannot have naysayer proofs in partial synchrony or asynchrony: if the adversary can arbitrarily delay the posting of naysayer proofs, then we cannot enforce soundness of the underlying proofs. Note that synchrony is already assumed by most of the deployed consensus algorithms, e.g., Nakamoto consensus [Nak08]. Furthermore, we assume that the public bulletin board offers censorship resistance for the writers of the bulletin board. Finally, we assume that there is *at least one honest party* who is ready to create and submit naysayer proofs for invalid proofs.

3.2 Formal Definitions

We formally introduce the notion of a *naysayer proof*, with the following syntax:

Definition 4 (Naysayer proof)**.** *Given a non-interactive proof system* $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ *for some NP relation* $\mathcal{R}$*, the corresponding naysayer proof system* Π_{nay} *is a tuple of PPT algorithms* $(\mathsf{NSetup}, \mathsf{NProve}, \mathsf{Naysay}, \mathsf{VrfyNay})$ *defined as follows:*

$\mathsf{NSetup}(1^\lambda, 1^\lambda_{\mathsf{nay}}) \to (\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}})$**:** *Given security parameters* 1^λ *and* 1^λ_{nay} *for* Π *and* Π_{nay}*, respectively, output common reference strings* crs *and* $\mathsf{crs}_{\mathsf{nay}}$*. This algorithm might use private randomness.*

$\mathsf{NProve}(\mathsf{crs}, x, w) \to \pi$**:** *Given a statement* x *and witness* w *such that* $(x, w) \in \mathcal{R}$*, compute* $\pi' \leftarrow \Pi.\mathsf{Prove}(\mathsf{crs}, x, w)$ *and* com *a commitment to the evaluation trace of* $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi')$*, output* $\pi := (\mathsf{com}, \pi')$*.*

$\mathsf{Naysay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \mathsf{aux}_{\mathsf{nay}}) \to \pi_{\mathsf{nay}}$**:** *Given a statement* x*,* $\pi = (\mathsf{com}, \pi')$ *where* π' *is a corresponding (potentially invalid) proof in proof system* Π*, and auxiliary information* $\mathsf{aux}_{\mathsf{nay}}$*, output a naysayer proof* π_{nay} *disputing* π'*.*

$\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \pi_{\mathsf{nay}}) \to \{0, \perp\}$**:** *Given a statement-proof pair* $(x, \pi = (\mathsf{com}, \pi'))$ *and a naysayer proof* π_{nay} *disputing* π*, output a bit indicating whether evidence against* π *is sufficient to reject* π *(0) or inconclusive (*$\perp$*).*

A trivial naysayer proof system always exists in which $\pi_{\mathsf{nay}} = \varnothing$, $\pi = (\perp, \pi')$, and $\mathsf{VrfyNay}$ simply runs the original verification procedure. We say a proof system Π is *efficiently naysayable* if there exists a corresponding naysayer proof system Π_{nay} such that $\mathsf{VrfyNay}$ is asymptotically faster than Vrfy. If $\mathsf{VrfyNay}$ is only concretely faster than Vrfy, we say Π_{nay} is a *weakly efficient* naysayer proof. Note that some proof systems already have constant proof size and verification time [Gro16, Sch90] and therefore can, at best, admit only weakly efficient naysayer proofs. Moreover, if $\mathsf{aux}_{\mathsf{nay}} = \varnothing$, we say Π_{nay} is a *public* naysayer proof.

Definition 5 (Naysayer correctness)**.** *Given a proof system* Π*, a naysayer proof system* Π_{nay} *is* correct *if, for all honestly generated* $\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}$*, all statements* x*, and all invalid proofs* π*,* Naysay *outputs a valid naysayer proof* π_{nay}*:*

$$\Pr\left[\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \pi_{\mathsf{nay}}) = 0 \;\middle|\; \begin{array}{c} (\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}) \leftarrow \mathsf{NSetup}(1^\lambda, 1^\lambda_{\mathsf{nay}}) \wedge \\ \mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 0 \wedge \\ \pi_{\mathsf{nay}} \leftarrow \mathsf{Naysay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \mathsf{aux}_{\mathsf{nay}}) \end{array}\right] = 1. \tag{1}$$

Definition 6 (Naysayer soundness)**.** *Given a proof system Π, a naysayer proof system Π_{nay} is sound if, for all PPT adversaries $\mathcal{A}$ and for all x, aux, honestly generated $\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}$, and correct proofs π, $\mathcal{A}$ produces a verifying naysayer proof π_{nay} with at most negligible probability:*

$$\Pr\left[\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \pi_{\mathsf{nay}}) = 0 \;\middle|\; \begin{array}{c} (\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}) \leftarrow \mathsf{NSetup}(1^{\lambda}, 1^{\lambda}_{\mathsf{nay}}) \wedge \\ \mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 1 \wedge \\ \pi_{\mathsf{nay}} \leftarrow \mathcal{A}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \mathsf{aux}_{\mathsf{nay}}) \end{array}\right] \leq \mathsf{negl}(\lambda_{\mathsf{nay}}). \tag{2}$$

We distinguish between two types of naysayer proofs as follows.

Type 1. A prover of an NP-relation $\mathcal{R}_{\mathcal{L}}$ posts (x, π) to the public bulletin board claiming that $x \in \mathcal{L}$. If the proof π is invalid with respect to the statement x, i.e., $\mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 0$, then naysayer provers convince the resource-constrained verifier by sending a π_{nay} that this is indeed the case, i.e., $\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi), \pi_{\mathsf{nay}}) = 0$.

Type 2. This family of naysayer proofs is even more efficient in the optimistic case, as the prover *only sends the instance* x and no proofs at all, claiming without evidence that $(x, w) \in \mathcal{R}$. On the other hand, if the prover's assertion is incorrect, i.e., $(x, w) \notin \mathcal{R}$, then a naysayer prover provides the correct statement x' such that $(x', w) \in \mathcal{R}$ and a corresponding "regular" proof π such that $\mathsf{Vrfy}(\mathsf{crs}, x', \pi) = 1$. For example, in the case of rollups, the (public) witness w is the set of transactions in the rollup, and the statement $x = (\mathsf{st}, \mathsf{st}')$ is the updated rollup state after applying the batch w. Therefore, an incorrect assertion represents an incorrect application of the update w. The correction x' is the result of the proper application of w.

We conjecture that in most applications, in the worst case, type-2 naysayer proofs are more costly than type-1 naysayer proofs (both compute and storage). It is an interesting open question which applications are more suited to type-1 or type-2 naysayer proofs considering both optimistic and pessimistic costs. To thoroughly model this question, one must take into account the verifier's compute cost, the (naysayer) proof storage costs, as well as the probability of the prover sending an invalid proof. We leave this problem to future work. In the rest of this paper, we solely focus on type-1 naysayer proofs.

Finally, we show that for every NP language, there exists a logarithmic-size naysayer proof with constant verification time (i.e., a succinct naysayer proof).

Theorem 1. *For every NP language $\mathcal{L}$ with relation $\mathcal{R}_{\mathcal{L}}$, there exists a naysayer proof system Π_{nay} with logarithmic-sized proofs π_{nay} and constant-time verifier.*

Proof. For any NP language $\mathcal{L}$, there exists a non-interactive proof system $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ for $\mathcal{R}_{\mathcal{L}}$ where $\mathsf{Vrfy}(\mathsf{crs}, \cdot, \cdot)$ can be represented as a boolean circuit C of size $\mathsf{poly}(|x|)$ [LS91]. Recall that for all $(x, w) \in \mathcal{R}_{\mathcal{L}}$ and $\mathsf{crs} \leftarrow \Pi.\mathsf{Setup}(1^{\lambda})$, by correctness, we have that if $\pi' \leftarrow \Pi.\mathsf{Prove}(\mathsf{crs}, x, w)$ then

$C(x, \pi') = 1$. Therefore, if there is some gate of C for which the wire assignment is inconsistent, then the proof π' is incorrect. To naysay, i.e., to show the incorrectness of π', the naysayer simply provides the index of the inconsistent gate. Recall that the verifier has access to the wire assignments of $C(x, \pi')$ as part of π (see Definition 4). The verifier then checks whether the wire assignments of $C(x, \pi')$ are consistent with a correct evaluation of the gate, which is a constant-time operation assuming constant-time indexing into the wire assignments. Furthermore, the naysayer proof consists only of the gate index, which is logarithmic in the circuit size, i.e., succinct. □

Corollary 1. *Every efficient proof system* Π *(i.e., with a polynomial-time verification algorithm) has a succinct naysayer proof.*

Proof. Given any proof system Π, one can represent the $\mathsf{Vrfy}(\mathsf{crs}, \cdot, \cdot)$ algorithm as a circuit and apply the above theorem to obtain a succinct naysayer proof. □

3.3 Applications

The naysayer proof paradigm is generally applicable for proof systems with multi-round amplification, repetitive structure (e.g., multiple bilinear pairing checks [GWC19]), or recursive reduction (e.g., Pietrzak's proof of exponentiation [Pie19]). In this section, we highlight three example constructions of naysayer proofs.

3.3.1 FRI Polynomial Commitment Scheme

The FRI polynomial commitment scheme [BBHR18] is used as a building block in many non-interactive proof systems, including STARKs [BCGT13]. Below, we describe only the parts of FRI relevant to our discussion. The FRI commitment to a polynomial $p(x) \in \mathbb{F}^{\leq d}[X]$ is the root of a Merkle tree with $\rho^{-1}d$ leaves. Each leaf is an evaluation of $p(x)$ on the set $L_0 \subset \mathbb{F}$, where $\rho^{-1}d = |L_0| \ll |\mathbb{F}|$, for $0 < \rho < 1$. We focus on the verifier's cost in the opening proof of the FRI polynomial commitment scheme as applied in the STARK IOP. Let δ be a parameter of the scheme such that $\delta \in (0, 1 - \sqrt{\rho})$. The prover sends the verifier $\log_2(|L_0|)$ messages. The FRI opening proof's verifier queries the prover's each message $\lambda / \log_2(1/(1-\delta))$ times to ensure $2^{-\lambda}$ soundness error. In each query, the verifier needs to check a Merkle-tree authentication path consisting of $\mathcal{O}(\log_2(\rho^{-1}d))$ hashes. Therefore, the overall STARK proof consists of $\mathcal{O}(\lambda \log_2(\rho^{-1}d)/\log_2(1/(1-\delta)))$ hashes.

The overall STARK proof is invalid if any of the individual Merkle proofs is invalid. Therefore a straightforward naysayer proof $\pi_{\mathsf{nay}}^{\mathsf{FRI}} = (i, z_i)$ need only point to the ith node in one of the Merkle proofs, where the hash values of the children nodes x, y and their parent node $z \neq H(x, y)$ do not match in one of the incorrect Merkle authentication paths. The naysayer verifier only needs to compute a single hash evaluation $H(x, y) = z_i$ and check $z_i \neq z$. Thus, the naysayer proof for FRI has constant-size and can be verified in constant-time.

3.3.2 Post-quantum Signature Schemes

With the advent of account abstraction [Eth23a], Ethereum users can define their own preferred digital signature schemes, including post-quantum signatures as recently standardized by NIST [BHK+19, DKL+18, PFH+22]. In all known schemes, post-quantum signatures or public keys are substantially larger than their classical counterparts. Since post-quantum signatures are generally expensive to verify on-chain, they are prime candidates for the naysayer proof paradigm.

CRYSTALS-Dilithium [DKL+18]. The verifier of this scheme checks that the following holds for signature $\sigma = (\mathbf{z}, c)$, public key $pk = (\mathbf{A}, \mathbf{t})$, and message M:

$$\forall i : \|z_i\|_\infty \leq C \wedge \mathbf{Az} - c\mathbf{t} = \mathbf{w} \wedge c = H(M||\mathbf{w}), \tag{3}$$

where C is a constant, $\mathbf{A} \in R_q^{k\times l}$, and $\mathbf{z}, \mathbf{t}, \mathbf{w} \in R_q^k$ for the polynomial ring $R_q := \mathbb{Z}_q[x]/(X^{256} + 1)$. Notice that the checks in Equation (3) are efficiently naysayable. In fact, the naysayer prover must show that the following holds:

$$\exists i : \|z_i\|_\infty > C \vee \mathbf{Az} - c\mathbf{t} \neq \mathbf{w} \vee c \neq H(M||\mathbf{w}). \tag{4}$$

If the first check fails, then the naysayer prover shows an index i for which the infinity norm of one of the polynomials in $\mathbf{z}$ is large. If the second check fails, then the naysayer prover can point to the ith row of the vector $\mathbf{w}$, where matrix-vector multiplication fails and verify only that row. Finally, if the last check fails, then the naysayer verifier just needs to recompute a single hash evaluation.

SPHINCS+ [BHK+19]. The signature verifier in SPHINCS+ checks several Merkle authentication proofs, requiring hundreds of hash evaluations. A constant-size and -time naysayer proof can be easily devised akin to the naysayer proof described in Section 3.3.1. The naysayer prover simply points to the hash evaluation in one of the Merkle-trees where the signature verification fails.

3.3.3 Verifiable Shuffles

Verifiable shuffles are applied in many (blockchain) applications such as single secret leader election algorithms [BEHG20], mix-nets [Cha81], cryptocurrency mixers [SNBB19], and e-voting [Adi08]. The state-of-the-art proof system for proving the correctness of a shuffle is due to Bayer and Groth [BG12]. Their proof system is computationally heavy to verify on-chain as the proof size is $\mathcal{O}(\sqrt{n})$ and verification time is $\mathcal{O}(n)$, where n is the number of shuffled elements.

Most shuffling protocols (of public keys, re-randomizable commitments, or ElGamal ciphertexts) admit a succinct naysayer proof if the naysayer knows at least one of the shuffled elements. Let us consider the simplest case of shuffling public keys. We want to prove membership in the following NP language:

$$\mathcal{R}_{perm} := \{(g^{w_i}, g^{r\cdot w_{\sigma(i)}})_{i=1}^n, g^r; \sigma, r | \ \forall i : w_i, r \in_R \mathbb{F}_p, g \in \mathbb{G}, \sigma \in_R \mathsf{Perm}(n)\}, \tag{5}$$

where $\mathsf{Perm}(n)$ is the set of all permutations $f : [n] \to [n]$. Suppose the naysayer knows that for $j \in [n]$, the prover did not correctly include $g^{r \cdot w_j}$ in the shuffle. The naysayer can prove this by showing that $(g, g^{w_j}, g^r, g^{r \cdot w_j}) \in \mathcal{R}_{DH} \wedge g^{r \cdot w_j} \notin (\cdot, g^{r \cdot w_{\sigma(i)}})_{i=1}^n$, where $\mathcal{R}_{DH}$ is the language of Diffie-Hellman tuples. One can show that a tuple is a Diffie-Hellman tuple with a proof of knowledge of discrete logarithm equality [CP93]. However, the naysayer must know the discrete logarithm w_j to produce such a proof. Unlike our previous examples, which were publicly naysayable, this is a privately naysayable proof since the naysayer algorithm takes auxiliary input w_j. With the right data structure for the permuted list (e.g., a hash table), both of the above conditions can be checked in constant-time with a constant-size naysayer proof, resulting in exponential savings compared to directly verifying the original Bayer-Groth shuffle proof.

3.3.4 Evaluation

We evaluate the asymptotic cost savings for the verifiers in the four examples discussed in Section 3.3. Note that naysayer proofs allow an exponential speedup for the verifier for verifiable shuffles and a logarithmic speedup for the FRI polynomial commitment opening proof verifier, see Table 2. For CRYSTALS-Dilithium, we can only claim weakly efficient naysayer proofs, as there is no asymptotic gap in the complexity in certain branches of the signature verification circuit and the naysayer prover algorithm, cf. Equations (3) and (4).

	FRI Opening	**CRYSTALS-D.**	**SPHINCS+**	**Shuffle proof**
π storage	$\mathcal{O}(\lambda \log^2(d))\mathbb{H}$	$\mathcal{O}(\lambda)\mathbb{F}$	$\mathcal{O}(\lambda)\mathbb{F}$	$\mathcal{O}(\sqrt{n})\mathbb{G}$
$\mathsf{Vrfy}(\pi)$ compute	$\mathcal{O}(\lambda \log^2(d))\mathbb{H}$	$\mathcal{O}(\lambda)\mathbb{F} + 1\mathbb{H}$	$\mathcal{O}(\lambda)\mathbb{H}$	$\mathcal{O}(n)\mathbb{G}$
π_{nay} storage	$1\mathbb{F}$	$1\mathbb{F} \vee 1\mathbb{F} \vee 1\mathbb{F}$	$1\mathbb{F}$	$2\mathbb{G} + 1\mathbb{F}$
$\mathsf{NVrfy}(\pi_{nay})$ compute	$1\mathbb{H}$	$\mathcal{O}(\lambda)\mathbb{F} \vee \mathcal{O}(\lambda)\mathbb{F} \vee 1\mathbb{H}$	$1\mathbb{H}$	$4\mathbb{G}$

Table 2: Cost savings of the naysayer paradigm for the example applications in Section 3.3. In FRI, let $deg(p(x)) = d$. For the Bayer-Groth shuffle argument [BG12], we consider n shuffled public keys (or ciphertexts). $\mathbb{F}, \mathbb{G}$ denotes field/group elements or field/group operations, respectively. $\mathbb{H}$ denotes hashing operations.

3.4 Storage Considerations

We assumed in our evaluation that the naysayer verifier can read the instance x, the original proof π, and the naysayer proof π_{nay} entirely. Note that in the pessimistic case, the verifier requires increased storage (for π_{nay}) but only needs to compute $\mathsf{VrfyNay}$ instead of Vrfy. A useful naysayer proof system should compensate for increased storage by considerably reducing verification costs.

In either case, this approach of storing all data on chain may not be sufficient in blockchain contexts where storage is typically very costly. Blockchains such as Ethereum differentiate costs between persistent storage (which we can call S_{per}) and "call data" (S_{call}), which is available only for one transaction and is significantly cheaper as a result. Verifiable computation proofs, for example, are usually stored in S_{call} with only the verification result persisted to S_{per}.

Some applications now use a third, even cheaper, tier of data storage, namely off-chain *data availability services* (S_{DA}), which promise to make data available off-chain but which on-chain contracts have no ability to read. Verifiable storage, an analog of verifiable computation, enables a verifier to store only a short commitment to a large vector [CF13, Mer88] or polynomial [KZG10], with an untrusted storage provider (S_{DA}) storing the full values. Individual data items (elements in a vector or evaluations of the polynomial) can be provided as needed to S_{call} or S_{per} with short proofs that they are correct with respect to the stored commitment.

This suggests an optimization for naysayer proofs in a blockchain context: the prover posts only a binding commitment $H(\pi)$, which the contract stores in S_{per}, while the actual proof π is stored in S_{DA}. We assume that potential naysayers can read π from S_{DA}. In the optimistic case, the full proof π is never written to the more-expensive S_{call} or S_{per}. In the pessimistic case, when naysaying is necessary, the naysayer must send openings of the erroneous proof elements to the verifier (in S_{call}). The verifier checks that these data elements are valid with respect to the on-chain commitment $H(\pi)$ stored in S_{per}. Note that in some naysayer proof systems which don't require reading all of π, even this pessimistic case will offer significant savings over storing all of π in S_{call}. An important future research direction is to investigate this optimized storage model's implications and implementation details.

4 On-chain Voting and Auctions

5 Hot-Cold Threshold Wallets

(Parts of this section are taken/adapted from [GGJ+24].)

In the cryptocurrency ecosystem, anyone who controls the signing key for some account can take actions on the user's behalf. This makes signing keys very valuable cryptographic material, since they can allow an attacker to transfer potentially large sums of money out of a user's account with no recourse. A plethora of solutions has emerged to safeguard these keys in the form of cryptocurrency wallets.

Wallets can be classified into two types: custodial and non-custodial solutions. In a custodial wallet, a client trusts some third party (the custodian) to store the signing key on their behalf and use it to authorize transactions at their request. This clearly requires a strong trust assumption but relieves the client of the need to securely store such a high-value secret. At the other end of the spectrum, non-custodial wallets require clients to store their own signing keys, e.g., on their machine or in a hardware wallet [AGKK19]. This can be a usability issue, since by definition there cannot be a recovery or reset mechanism in case the client loses the key file or hardware wallet, or if the key material is compromised. An increasingly common compromise between these two extremes are threshold wallets [KMOS21, DEF$^+$23, BMP22, Eya21]. These are essentially custodial wallets with multiple custodians, where each one is given only a share of the signing key. This maintains the usability advantages of custodial wallets while reducing the trust placed in any single custodian. Unfortunately, the normally highly-interactive threshold signing procedures require custodians to be online during the signing phase, making them vulnerable to attacks aiming to learn their shares [**?**, **?**].

Cold wallets A common mitigation is to keep only limited funds in the online wallet, with most of a user's assets kept in a highly secure air-gapped wallet. This can be realized via, e.g., a deterministic wallet [**?**, DFL19, ADE$^+$20, Hu23, ER22], which enables unlinkable transfers to an offline wallet by specifying how to deterministically derive session keys from a master public-private key pair. Thus, the online wallet can compute and publish the current session public key, allowing anyone to transfer money to the cold wallet (which can derive the corresponding session private key).

This idea is standardized by the BIP32 proposal [Wui12], which also defines how these session keys can be derived in a hierarchical manner to create "child" wallets who can control their own funds. Because the master secret key (and any session/child secret keys) must still be stored in a trusted place, this makes the offline wallet a highly valuable target when it inevitably comes online to transfer funds to the online wallet. Although a recent work [DEF$^+$23] shows how to enable threshold child wallets, the issue of a single point of failure still persists at the root wallet.

Ensuring availability Another problem arises when the responsibility to store key material is delegated solely to external custodians who have no explicit incentive to make sure the key stays available: custodians may be lazy and stop storing the key material as a way to save space and money, relying on the assumption that the other custodians will act honestly and store their corresponding shares. Even in the case in which all custodians have honest intentions, a client with a large sum of money in their threshold wallet may still wish to intermittently check that a custodian has access to its key material. To our knowledge, no wallet explicitly implements such a feature, although a similar effect can be achieved by requesting approval of an empty transaction, since this makes it clear that at least t of the parties still hold their shares (although not necessarily which ones). A more targeted audit could be implemented via generic zero-knowledge proofs requiring a party to prove knowledge of the ith key share using some time-based challenge, but the practical efficiency of such a proof is unclear.

5.1 Our Results

We introduce a new type of wallet which combines offline components with the threshold wallet approach to achieve stronger security guarantees. In our model, each threshold signer consists of two parties: an online *hot* storage and an offline, resource-constrained *cold* storage, e.g., a secure enclave, trusted execution environment, or perhaps something even more sophisticated. Producing a threshold signature on any message should require the involvement of both the hot and cold parties while minimizing the computation and communication on the part of the cold party. Wallet owners should also be able to request proofs from any party, hot or cold, that it retains its share of the secret key. Finally, it should be possible to periodically refresh the shares of the online parties to thwart an adversary who slowly and incrementally corrupts parties in the system. We describe each of our contributions in more detail below.

Stronger threat model In a hot-cold threshold wallet, an attacker must corrupt *both* the cold and hot components of a signer to obtain its key share. This requirement to corrupt a threshold t of hot-cold *pairs* is different from a generic $2t$-out-of-$2n$ threshold wallet, since the attacker cannot forge a signature by corrupting any arbitrary set of $2t$ parties: in our model, corrupting, e.g., $2t$ hot parties should be of no use in forging a signature (in fact, even corrupting all n hot parties should be useless). Indeed, our threat model is instead comparable to a Boolean signing policy which requires at least t pairs of hot *and* cold parties to contribute, which is much more difficult to attack since the cold components are almost always offline.

Threshold BLS construction We show how to construct such a hot-cold threshold signing protocol for the BLS signature scheme [BLS01, Bol03]. Although ECDSA [GGN16, DKLs18, LN18, GG18, AHS20, GKSŚ20, DJN+20, CGG+20, CCL+20, CCL+21, Pet21, ANO+22] and Schnorr [KG20, BCK+22, BLM22] are the most popular threshold signature schemes in the literature, their more complicated structure makes a threshold signing procedure with offline parties difficult. BLS signatures have the advantage of being simple to understand and deploy. This has led to their widespread use in production systems, including in Ethereum's consensus protocol [Edg23, §2.9.1], Filecoin [?], transactions on the Chia Network [?], and a BLS smart contract wallet [?]. With the event of account abstraction on Ethereum [?], users will be able to specify alternative signature schemes to verify their transactions, further easing the adoption of BLS-based wallets. The homomorphic structure of BLS allows us to give a simple non-interactive hot-cold threshold signature protocol. Our construction also has the added benefit that the cold parties can generate their secret key shares independently, without interacting with the hot parties or with each other.

Proofs of remembrance We show how the cold and hot parties in our protocol can produce "proofs of remembrance", i.e., zero-knowledge proofs that they still possess their key material. In order to support proactive refreshes (see below), our proof systems must accommodate periodic updates to individual key shares while guaranteeing that the underlying secret key is still available. We show how to meet both needs simultaneously, allowing a client to intermittently and independently audit its (hot or cold) custodians. This allows it to ensure that its key material has not been overwritten or forgotten, and its funds are still accessible.

Proactive refresh Many threshold wallets support proactive refresh [KMOS21], meaning secret key shares are regularly updated. This forces an attacker to compromise at least t parties within a single epoch in order to mount a successful attack: otherwise, any partial key material which has been obtained is made obsolete by the refresh operation. Our protocol allows proactive refresh of the hot key shares while still letting hot parties prove knowledge of those shares and their consistency with the (unchanging) verification key.

Efficient open-source implementation We implemented our protocol and show that it is practically efficient for essentially all reasonable settings of t, n. Producing a signature takes less than 1ms and computing proofs of remembrance is on the order of milliseconds. Our implementation is publicly available in Hyperledger Labs[5] and is Apache 2.0 licensed.

5.2 Technical overview

We will use the superscripts hot and cold, respectively, to denote the hot and cold components of some value, and a subscript i to denote the value corresponds to the ith party. For example, the ith party's cold signature is written as σ_i^{cold}. We use the words "user" and "client" interchangeably to refer to the wallet owner.

A starting point for constructing a threshold signature with our desired access structure would be to first compute a t-out-of-n sharing $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ of the signing key sk, then give each hot-cold pair a 2-out-of-2 (additive) sharing $\mathsf{sk}_i^{\mathsf{hot}}, \mathsf{sk}_i^{\mathsf{cold}}$ of one of the threshold shares. The signing protocol has to be designed carefully, since we need to ensure that any communication from the cold party to the hot is "bound" to the message m being signed. Otherwise, the hot storage could replay the communication and produce a signature on some other message $m' \neq m$ without the cold storage's cooperation, violating our threat model.

Our scheme takes advantage of the natural homomorphism of BLS signatures to use the additive sharing idea. At a high level, the cold and hot parties will receive shares r and $\mathsf{sk}_i + r$, respectively, of the threshold signing key share. Then the cold storage can send the hot storage a partial signature $\sigma_i^{\mathsf{cold}} := H(m)^r$ which is "bound" to the message m, and the hot storage computes its own partial signature $\sigma_i^{\mathsf{hot}} := H(m)^{\mathsf{sk}_i + r}$. The threshold BLS signature σ_i can be computed by the hot storage as $\sigma_i^{\mathsf{hot}} / \sigma_i^{\mathsf{cold}} = H(m)^{\mathsf{sk}_i}$. This scheme also enables

[5] https://github.com/hyperledger-labs/agora-key-share-proofs

proactive refresh of the hot shares basically "for free" due to the malleability of the additive sharing: the wallet owner can simply send every hot storage a t-out-of-n sharing of zero, which the hot storage adds to its current share.

Our idea for realizing this additive sharing is to allow each cold party to sample an encryption key-pair $(\mathsf{ek}_i, \mathsf{dk}_i)$ and let the cold share r be a deterministic function of ek_i and the client signing key sk. In particular, we want to set $r := \mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}})$ for a carefully-designed efficient hash function $\mathcal{H}$. Put another way, this means the hot key share will be an encryption of sk_i under the cold party's public key ek_i. This allows the hot key material to be computed by a key generation protocol which outputs the verification key $\mathsf{vk} = g^{\mathsf{sk}}$ and each hot party's key share $\mathsf{sk}_i + \mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}})$ without interacting with the cold parties. The crucial piece is that each cold encryption key-pair will be instantiated in the same group and using the same generator as the signing key-pair, so $\mathsf{ek}_i^{\mathsf{sk}} = (g^{\mathsf{dk}_i})^{\mathsf{sk}} = \mathsf{vk}^{\mathsf{dk}_i}$. Thus, given the verification key, each cold storage can now independently recover its signing key share r as $\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})$. This means that a partial signature is effectively computed by decrypting the hot key share "in the exponent" of the partial BLS signature.

To prove remembrance, each type of party uses its own proof system. For cold parties, a standard proof of knowledge of discrete logarithm (for dk_i corresponding to ek_i) suffices. For the hot parties, the relation is more complicated since the shares are no longer simply points on a degree-t polynomial. We give a proof system based on KZG commitments which leverages their multiplicative homomorphism to enable compatibility with hot share refreshes.

5.3 Related Work

Blokh et al. [BMP22] describe another threshold signing protocol which combines online (hot) and offline (cold) parties. Their protocol is tailored to ECDSA signatures and uses n hot parties and single cold party. These $(n+1)$ parties are independent (i.e., the cold party is not paired with any hot party, as in our setting); thus, their security guarantee is still a classic t-out-of-n threshold guarantee, unlike our protocol, which is secure up to corruption of t *pairs*. Furthermore, in their protocol, hot parties engage in an MPC to generate a pre-signature, which is then finalized and output by the cold party. This is in contrast to our protocol, where the cold parties send messages to their corresponding hot parties, and the hot parties output the final signature (shares).

5.4 Model

Let $I_1, \ldots, I_n$ be parties (each representing an institution) who will store shares of some user's signing key sk. Each institution I_i controls two parties: a hot storage (P_i^{hot}) and a cold storage (P_i^{cold}). Thus we represent an institution by the tuple $I_i = (P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})$. As in the standard threshold wallet setting, the hot parties are connected to each other via authenticated (but not private) channels and we also assume the parties can send broadcast messages (e.g. by posting to a blockchain). In contrast, each cold storage is connected by an

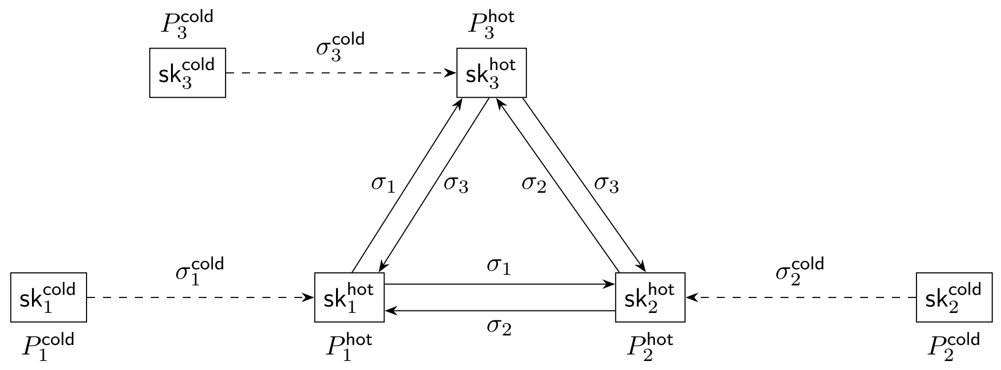
P_3^{cold}
$\mathsf{sk}_3^{\mathsf{cold}}$
σ_3^{cold}
P_3^{hot}
$\mathsf{sk}_3^{\mathsf{hot}}$
σ_1
σ_3
σ_2
σ_3
$\mathsf{sk}_1^{\mathsf{cold}}$
σ_1^{cold}
$\mathsf{sk}_1^{\mathsf{hot}}$
σ_1
$\mathsf{sk}_2^{\mathsf{hot}}$
σ_2^{cold}
$\mathsf{sk}_2^{\mathsf{cold}}$
σ_2
P_1^{cold}
P_1^{hot}
P_2^{hot}
P_2^{cold}

Figure 2: A hot-cold threshold wallet with $n = 3$. Given a message m, each P_i^{cold} uses its cold share $\mathsf{sk}_i^{\mathsf{cold}}$ to compute a cold partial signature σ_i^{cold} on m and sends it to its hot storage. P_i^{hot} uses m and its hot share $\mathsf{sk}_i^{\mathsf{hot}}$ to compute σ_i^{hot}, which it combines with σ_i^{cold} to get σ_i. The hot parties broadcast their partial signatures to each other to reconstruct the full signature σ on m. The dashed lines between each cold and hot storage represent the authenticated channel between them which is only active at signing time. The hot parties are always online and connected, represented by the solid lines.

authenticated channel only to its corresponding hot storage. The channel is only active during the signing phase, further reducing the cold party's attack surface. (For example, the cold storage could be a read-only USB device which is plugged into a PC (the hot storage) only briefly to produce a signature.) An illustration of this model is given in Figure 2 for $n = 3$.

We assume P_i^{hot} has more storage space and computational power, while P_i^{cold} has limited storage (in particular, we want the space complexity to be independent of the number of users in the system). Therefore, our protocol aims to minimize the computation done by the cold storage.

5.5 Malleable Encryption for eXponents (MEX)

As explained above, the core idea of our construction is for each hot party to store an encryption of the institution's signing key share sk_i, with the corresponding decryption key kept on the cold storage. That is, each institution will generate an encryption key pair $(\mathsf{ek}_i, \mathsf{dk}_i)$ and store the hot key share $\mathsf{sk}_i^{\mathsf{hot}} := \mathsf{Enc}(\mathsf{ek}_i, \mathsf{sk}_i)$ in the hot storage and $\mathsf{sk}_i^{\mathsf{cold}} := \mathsf{dk}_i$ in the cold storage. We want to enable threshold signing of a message m by allowing the hot and cold parties to derive signature shares $\sigma_i^{\mathsf{hot}}, \sigma_i^{\mathsf{cold}}$ for m from their secret material $\mathsf{sk}_i^{\mathsf{hot}}, \mathsf{sk}_i^{\mathsf{cold}}$ (the secret key and ciphertext, respectively). Together, the signature shares can be used to recover a partial signature σ_i of m under sk_i.

In this section, we describe an encryption scheme based on ElGamal [ElG84] whose malleability allows decryption of the hot key share "in the exponent" of a partial BLS signature. Recall that the ElGamal ciphertext for a message $m \in \mathbb{G}$ is computed as $(g^r, \mathsf{ek}^r \cdot m) \in \mathbb{G}^2$ where $\mathbb{G}$ is a prime order group. In our case, however, $m = \mathsf{sk}_i$ will be a scalar element of $\mathbb{Z}_p$ (an "exponent"). Adapting the encryption scheme to scalar messages, we can compute the ciphertext as $(g_2^r, \mathcal{H}(\mathsf{ek}^r) + m)$, where $\mathcal{H}$ is some function which maps $\mathsf{ek}^{\mathsf{sk}}$ to $\mathbb{Z}_p$ in a way that masks m. We will see below that this can be achieved by defining our encryption key ek to be a tuple $(\mathsf{ek}_1, \mathsf{ek}_2) \in \mathbb{G}^2$ and letting $\mathcal{H} : \mathbb{G}^2 \to \mathbb{Z}_p$. Now we can mask m with the output of $\mathcal{H}$, and a cold party can undo this masking (in the exponent of the BLS signature) using g^r and $\mathsf{dk}_1, \mathsf{dk}_2$. We call the resulting scheme Malleable Encryption for eXponents (MEX).

Like the original ElGamal encryption and as the name suggests, MEX is malleable (in this case, additively rather than multiplicatively), which allows "shifting" of the message m in a ciphertext by an additive factor (via the Shift algorithm). We will use this property to enable proactive refresh of the hot (encrypted) key shares.

Construction 5 (MEX)**.** *Let $\mathbb{G}$ be a DLog-hard group of order p with generator g. Let $\mathcal{H} : \mathbb{G} \to \mathbb{Z}_p$ be a hash function.*

- $\underline{(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KGen}(1^\lambda)}$*: Sample* $\mathsf{dk} \xleftarrow{\$} \mathbb{Z}_p$*. Set* $\mathsf{ek} := g^{\mathsf{dk}}$ *and output* $(\mathsf{ek}, \mathsf{dk})$*.*
- $\underline{ct \leftarrow \mathsf{Enc}(\mathsf{ek}, m; r)}$*: Given an encryption key* $\mathsf{ek} \in \mathbb{G}$ *and a message* $m \in \mathbb{Z}_p$*, use randomness* $r \in \mathbb{Z}_p$ *to compute the ciphertext* $ct := (g^r, m + \mathcal{H}(\mathsf{ek}^r))$*.*
- $\underline{m' \leftarrow \mathsf{Dec}(\mathsf{dk}, ct)}$*: Given a secret key* $\mathsf{dk} \in \mathbb{Z}_p$ *and a ciphertext* $ct \in \mathbb{G} \times \mathbb{Z}_p$*, parse* ct *as* (ct_0, ct_1) *and return* $ct_1 - \mathcal{H}(ct_0^{\mathsf{dk}})$*.*
- $\underline{ct' \leftarrow \mathsf{Shift}(ct, \delta)}$*: Given a ciphertext* $ct \in \mathbb{G} \times \mathbb{Z}_p$ *and a shift* $\delta \in \mathbb{Z}_p$*, parse* ct *as* (ct_0, ct_1) *and output the shifted ciphertext* $ct' := (ct_0, ct_1 + \delta)$*.*

5.5.1 Additive Secret Sharing from MEX

The Enc algorithm in Construction 5 takes an explicit randomness input r. In the context of our wallet construction, encryption will take place at the same time as signing key generation, and instead of sampling fresh randomness $r \in \mathbb{Z}_p$ for each hot party's ciphertext (hot key share), we will sample a single value r. The signing key-pair is set to $(\mathsf{vk}, \mathsf{sk}) := (g^r, r)$.[6] The result is that each party P_i^{hot}'s ciphertext uses the mask $\mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}}) = \mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})$, which allows the corresponding cold party P_i^{cold} to decrypt using only a client's verification key vk and without receiving or storing any additional per-client randomness.

Because the input to $\mathcal{H}$ is no longer uniformly random, we now need $\mathcal{H}$ to be a hash function where the Leftover Hash Lemma (see Section 2.7) holds on random inputs. In order for the output of $\mathcal{H}$ to have sufficient entropy to

[6]This means we can leave out the redundant first element, so each ciphertext will consist of a single group element.

mask m, this requires two group elements as input. Therefore, in our wallet construction we will use $\mathsf{ek} := (g^{\mathsf{dk}_1}, g^{\mathsf{dk}_2})$ and $ct := m + \mathcal{H}(\mathsf{ek}_1^{\mathsf{sk}}, \mathsf{ek}_2^{\mathsf{sk}})$.

Although any function $\mathcal{H}$ which meets the above requirements suffices, it is desirable to find a very efficient construction since $\mathcal{H}$ will have to be computed by the cold party at signing time. One such $\mathcal{H}$ is a random subset sum. In more detail, $\mathcal{H}$ first represents its inputs $x_1, x_2 \in \mathbb{G}$ as ℓ-bit vectors $\mathbf{x}_1, \mathbf{x}_2 \in \{0,1\}^\ell$, where $\ell = \log p$. Let $\mathbf{r}_1, \mathbf{r}_2 \in \mathbb{Z}_p^{2\ell}$ be (public) uniform vectors with $\mathbf{r}_k = (\mathbf{r}_{k,0}, \mathbf{r}_{k,1})$ for $k = 1, 2$. We will use bracket notation to index into the vector, i.e., $\mathbf{r}_{1,0}[i]$ is the ith element of $\mathbf{r}_{1,0}$. Let $\mathcal{H}(x_1, x_2) := \mathcal{H}'(\mathbf{r}_1, \mathbf{x}_1) + \mathcal{H}'(\mathbf{r}_2, \mathbf{x}_2)$, where $\mathcal{H}' : \mathbb{Z}_p^{2\ell} \times \{0,1\}^\ell \to \mathbb{Z}_p$ is the subset sum function

$$\mathcal{H}'(\mathbf{r} := (\mathbf{r}_0, \mathbf{r}_1), \mathbf{x}) := \sum_{b_i \in \mathbf{x}} \mathbf{r}_{b_i}[i]$$

When we want to be specific about the randomness used in $\mathcal{H}$, we write $\mathcal{H}(x_1, x_2; \mathbf{k})$ for $\mathbf{k} \in \mathbb{Z}_p^{4\ell}$. By Lemma 1 in Section 2.7, the output of $\mathcal{H}$ is statistically indistinguishable from uniform.

5.6 Proofs of Remembrance

We will use non-replayable zero-knowledge proofs of knowledge (ZKPoKs) for two languages. We refer the reader to [?] for the definition of a ZKPoK. The PoKs for both our hot and cold parties are Sigma protocols, made non-interactive via the Fiat-Shamir transform [FS87]. Non-replayability is enforced by including some unpredictable timestamp (e.g., current block number of some blockchain) in the payload of the Fiat-Shamir hash.

To prove knowledge of the cold share, i.e., its decryption key, each cold storages will use a proof of knowledge of discrete logarithm Π_{DL} for the language $\{y \in \mathbb{G} : \exists w \in \mathbb{Z}_p \text{ s.t. } y = g^w\}$. This can be instantiated with the classic Schnorr protocol [Sch90].

The proof of knowledge for the hot storages is more complicated. At setup time, each hot storage must receive, along with its encrypted share, a proof of knowledge for the share's well-formedness. This should prove that the hot share equals $\mathcal{H}(\mathsf{ek}_{i,1}^x, \mathsf{ek}_{i,2}^x) + x_i$ for some secret value x such that $g^x = \mathsf{vk}$ and $x_i = \mathsf{Share}(x, t, n)$, and that the same value of x is used for every party. Our core idea is to use a KZG commitment to the polynomial used in the Shamir sharing of x and distribute an evaluation proof to each hot storage. The additive homomorphism of the commitments allows the public commitment, as well as each party's evaluation proof, to be updated when the shares are refreshed. We begin with a strawman example where the hot storage shares are unencrypted and then show how to adapt the construction to encrypted shares.

5.6.1 PoK of (Unencrypted) Key Share

Let crs be a degree-$(t-1)$ KZG CRS. At time $T = 0$, the client C picks a random degree-$(t-1)$ polynomial $f_0(X) \in \mathbb{Z}_p[X]$ (its evaluation at 0 will be the signing key sk) and publishes $\mathsf{com}_0 \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, f_0(X))$. Each hot storage P_i^{hot} receives an opening $f_0(i)$ (i.e., the key share sk_i) and evaluation proof $\pi_{0,i}$.

Whenever it wants the servers to refresh their shares and thus transition from epoch $T-1$ to T, C will commit to (as ucom_T) a new random degree-t polynomial $z_T(X)$ such that $z_T(0) = 0$, publish an evaluation proof $\zeta_{T,0}$ at $X = 0$, and send each P_i^{hot} its opening $z_T(i)$ and the corresponding evaluation proof $\zeta_{T,i}$. Everyone can check the correctness of the update by verifying $\zeta_{T,0}$ with respect to ucom_T. If the check passes, they can compute the commitment to the new polynomial $f_T(X)$ homomorphically from the commitments to $f_{T-1}(X)$ and $z_T(X)$, namely as $\mathsf{com}_T = \mathsf{com}_{T-1} \cdot \mathsf{ucom}_T$. Then each party verifies its own update proof $\zeta_{T,i}$ before updating its previous key share $f_{T-1}(i)$ and proof $\pi_{T-1,i}$, respectively, to $f_T(i) := f_{T-1}(i) + z_T(i)$ and $\pi_{T,i} := \pi_{T-1,i} \cdot \zeta_{T,i}$. By the homomorphic nature of the KZG commitment scheme, P_i^{hot} now has an evaluation of $(f_{T-1} + z_T)(X) = f_T(X)$ at i and a corresponding evaluation proof $\pi_{T,i}$.

There is a problem with this scheme: P_i^{hot} needs to reveal $f_T(i)$ in order for anyone to verify $\pi_{T,i}$...but $f_T(i)$ is its current key share! The solution is for P_i^{hot} to prove it knows $f_T(i)$ in *zero-knowledge* via a blinded KZG evaluation proof (a modified version of the protocol in [ZBK+22, §6.1]). It commits to the key share $f_T(i)$ using a Pedersen commitment $\mathsf{com}_{\mathsf{ped}} := g_1^{f_T(i)} h_1^r$ and computes $\pi_{\mathsf{ped}} \leftarrow \Pi_{\mathsf{ped}}.\mathsf{Prove}(\mathsf{com}_{\mathsf{ped}}; (f_T(i), r))$. It also samples $s \xleftarrow{\$} \mathbb{Z}_p$ and computes a blinded version of the evaluation proof as $\overline{\pi}_{T,i} := \pi_{T,i} h_1^s$. The final ZKPoK is defined as $(\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}, \overline{\pi}_{T,i}, g^{s_{T,i}(\tau)})$, where $s_{T,i}(X) := -r - s(X - i)$. The client accepts if and only if

$$e(\mathsf{com}_T / \boxed{\mathsf{com}_{\mathsf{ped}}}, g_2) \overset{?}{=} e(\boxed{\overline{\pi}_{T,i}}, g_2^{\tau}/g_2^{i}) \cdot \boxed{e(h_1, g_2^{s_{T,i}(\tau)})}$$

(where the $\boxed{\text{boxed}}$ parts are changes to the original KZG verification check due to the blinding).

5.6.2 PoK of Encrypted Key Share

Next, we modify the previous construction to accommodate an *encrypted* initial key share $\tilde{x}_i := \mathcal{H}(\mathsf{ek}_{i,1}^{x}, \mathsf{ek}_{i,2}^{x}) + x_i$, where $x = f_0(0)$ and $x_i = f_0(i)$. After having computed every party's plaintext share x_i (as above), the client C will compute each encrypted key share $\tilde{x}_i$ and interpolate the degree-$(n-1)$ polynomial $\tilde{f}_0(X)$ where $\tilde{f}_0(i) = \tilde{x}_i$ for $i \in [n]$. For now, we assume the $\tilde{x}_i$ and $\tilde{f}_0(X)$ are computed correctly (we will return to this assumption in Section 5.7). Thus, we don't prove the correctness of the $\tilde{x}_i$ values themselves, only that they are equal to $\tilde{f}_0(i)$ where $\tilde{f}_0(X)$ is fixed for all parties.

C commits to $\tilde{f}_0(X)$ publicly and, in a similar fashion as before, sends each hot storage P_i^{hot} the opening $\tilde{f}_0(i) =: \tilde{x}_i$ and the corresponding evaluation proof $\pi_{0,i}$. Now the hot storage can prove rembrance of its current share $\tilde{f}_T(i)$ in zero-knowledge in the same way as before, namely by committing to the share and blinding the evaluation proof $\pi_{T,i}$ as described in the previous section.

Hot Storage Proofs of Encrypted Key Share (Π_{EKS})

Parameters: Generators $g_1, h_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$; a degree-d KZG common reference string $\mathsf{crs} = \{g_1, g_1^\tau, \ldots, g_1^{\tau^d}, g_2, g_2^\tau\}$.

$\underline{\mathsf{Prove}((\mathsf{crs}, \mathsf{com}_T, i); (\tilde{x}_i, \pi_{T,i})) \to \pi_i^{\mathsf{hot}}}$: Given crs, a KZG commitment com_T to the current polynomial $\tilde{f}_T(X)$, and its index i, a hot storage uses its key share $\tilde{x}_i = \tilde{f}_T(i)$ and corresponding opening proof $\pi_{T,i}$ to compute a ZKPoK of $\tilde{x}_i$ as follows:

1. Sample $r \xleftarrow{\$} \mathbb{Z}_p$, let $\mathsf{com}_{\mathsf{ped}} := g_1^{\tilde{x}_i} h_1^r$, and compute $\pi_{\mathsf{ped}} \leftarrow \Pi_{\mathsf{ped}}.\mathsf{Prove}(\mathsf{com}_{\mathsf{ped}}; (\tilde{x}_i, r))$ (see Section 2.6).
2. Sample $s \xleftarrow{\$} \mathbb{Z}_p$ and let $s_i(X) := -r - s(X - i)$. Compute $\overline{\pi}_{T,i} := \pi_{T,i} h_1^s$ and $S := g_2^{s_i(\tau)}$ using crs.
3. Output $\pi_i^{\mathsf{hot}} := (\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}, \overline{\pi}_{T,i}, S)$.

$\underline{\mathsf{Vrfy}((\mathsf{crs}, \mathsf{com}_T, i), \pi_i^{\mathsf{hot}}) \to \{0,1\}}$: Given crs, a KZG commitment com_T, and a party index i, verify the hot storage proof $\pi_i^{\mathsf{hot}} = (\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}, \overline{\pi}_{T,i}, S)$ by outputting 1 iff the following hold:

$$\Pi_{\mathsf{ped}}.\mathsf{Vrfy}(\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}) = 1$$
$$e(\mathsf{com}_T / \mathsf{com}_{\mathsf{ped}}, g_2) = e(\overline{\pi}_{T,i}, g_2^\tau / g_2^i) \cdot e(h_1, S).$$

Figure 3: The proof system Π_{EKS} used by each P_i^{hot} to show possession of a valid encrypted key share with respect to the current KZG commitment.

Share refreshes in epoch T also proceed as before with a commitment ucom_T to a polynomial $z_T(X)$ such that $z_T(0) = 0$ (confirmed via a public evaluation proof $\zeta_{T,0}$ at $X = 0$). Parties receive their update value $z_T(i)$ and corresponding evaluation proof $\zeta_{T,i}$, and can update their (now encrypted) share homomorphically just like before. (This still works because our encryption scheme allows additive shifts of the plaintext via addition to the ciphertext.) The only difference is that, because the encrypted shares now lie on a degree-$(n-1)$ polynomial instead of a degree-$(t-1)$ polynomial, the KZG CRS must accomodate polynomials up to degree $n-1$. (In practice, because different clients in the system may chose different values of n, the KZG CRS will actually have some degree d which is as large as the maximum allowed value of $n-1$.) As with the original polynomial $\tilde{f}_0(X)$, we assume $z_T(X)$ is chosen honestly by C. In Section 5.8.1, we show how to avoid trusting C in the refresh stage.

The final hot proof of remembrance is summarized as Π_{EKS} in Figure 3.

5.7 Hot-Cold Threshold BLS

In this section, we show how to use the MEX from Section 5.5 and the proofs of remembrance from Section 5.6 to construct a hot-cold threshold wallet for BLS signatures (recalled in Section 2.4) with proofs of remembrance and proactive refresh of the hot shares.

Let $\mathbb{G}_1, \mathbb{G}_2$ be elliptic curve groups of order p generated by g_1 and g_2, respectively, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an efficiently computable asymmetric (type-3) pairing between them. Since $\mathsf{vk} \in \mathbb{G}_2$, we will also instantiate the MEX over $\mathbb{G}_2$. Let $H : \{0,1\}^* \to \mathbb{G}_1$ and $\mathcal{H} : \mathbb{G}_2^2 \to \mathbb{Z}_p$ be hash functions as defined in Section 2.4 and Section 5.5, respectively.

Subprotocols We assume the existence of the following ideal functionalities, which we will use as building blocks for our protocol:

(Encrypted) secret share generation $\mathcal{F}_{\mathsf{SS}}$: Presented in Figure 4, this functionality is executed between a client C and a set of institutional entities. The functionality allows the client to choose which institutional entities it wants to use. We require that the client provide the public keys of the institutional servers that it wants to engage. A direct way to implement the $\mathcal{F}_{\mathsf{SS}}$ functionality would be for (trusted) C to execute the steps of $\mathcal{F}_{\mathsf{SS}}$ on input $\mathsf{SSSetup}$ locally and send $\tilde{x}_i$ to P_i^{hot} for each i via a point-to-point channel. While this suffices for a number of applications, we model it as an abstract ideal functionality as we will be interested in settings where security is desired even if C (i) does not have access to a source of true randomness, or (ii) is (or, may in future be) corrupted. In Section 5.8.1, we show that one can use an additional proof system Π_{Ref} and a public bulletin-board with limited programmability to avoid trusting C beyond the key generation phase. We leave a fully decentralized key generation protocol to future work.

Public key infrastructure $\mathcal{F}_{\mathsf{PK}}$: Finally, we assume a functionality $\mathcal{F}_{\mathsf{PK}}$ (Figure 5) which allows a party (institutional cold servers in our case) to obtain a secret (decryption) key dk sampled from $\mathbb{Z}_p$ while its public (encryption) key $\mathsf{ek} := g_2^{\mathsf{dk}}$ is made public, and can be retrieved reliably by any client. Again this functionality can be implemented by a party executing it locally. We abstract it out to separate the implementation details of this functionality from our modeling.

Construction Our protocol for BLS signatures is given in Figures 6 and 7. Each cold storage is set up separately and independently of any client using the $\mathsf{ColdRegister}$ protocol. When a client registers, it specifies a set of n institutions $\mathcal{I}$ and a threshold $t \leq n$ of them required for signing. The $\mathsf{ClientRegister}$ protocol is an interactive protocol between C and the n hot parties specified by $\mathcal{I}$ which outputs a verification key vk to the client and an encrypted secret key share $\tilde{x}_i$ to each hot party. Each hot party also receives a proof π_i which will allow it to prove that $\tilde{x}_i$ was the output of $\mathsf{ClientRegister}$.

Public parameters: Groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p with generators g_1, g_2, respectively; a degree-d KZG common reference string crs.

- On input $(\mathtt{sid}, \mathsf{SSSetup}, C, (t, \mathcal{P}, \{\mathsf{ek}_i\}_{i\in[n]}))$, where $\mathcal{P} = \{P_1, \ldots, P_n\}$ is a set of parties, for $i \in [n]$, $\mathsf{ek}_i \in \mathbb{G}_2$ is an encryption key, and $t \leq |\mathcal{P}|$, it proceeds as follows:
  1. Sample $x \xleftarrow{\$} \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
  2. Generate t-out-of-n Shamir Shares of x as $x_1, \ldots x_n \in \mathbb{Z}_p$. Let $y_i := g_2^{x_i}$ $\forall i \in [n]$.
  3. Interpolate the degree-n polynomial $\tilde{f}$ such that $\tilde{f}(i) = \mathcal{H}(\mathsf{ek}_i^x) + x_i$ $\forall i \in [n]$. Compute $\mathsf{com} \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, \tilde{f})$.
  4. Delete any entries $(C, *, *, *, *, *) \in D$. Add $(C, \mathcal{P}, t, y, \mathsf{com}, \mathtt{time} := 1)$ to D.
  5. For each $i \in [n]$, compute $(\tilde{x}_i, \pi_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, \tilde{f}, i)$ and output $(\mathtt{sid}, \mathsf{SecretShare}, P_i, (C, i, \tilde{x}_i, \pi_i))$.
  6. Finally, output $(\mathtt{sid}, \mathsf{SSSetupDone}, C, (y, \{y_i\}_{i\in[n]}))$.
- On input $(\mathtt{sid}, \mathsf{ZeroSetup}, C, (t, \mathcal{P}))$, where $\mathcal{P} = \{P_1, \ldots, P_n\}$ is a set of parties and $t \leq |\mathcal{P}|$, it proceeds as follows:
  1. Generate t-out-of-n Shamir Shares of 0 as $x_1, \ldots x_n \in \mathbb{Z}_p$; let f be the polynomial used.
  2. Compute $\mathsf{com}_0 \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, f)$.
  3. Retrieve $(C, \mathcal{P}, t, y, \mathsf{com}, \mathtt{time}) \in D$ for the maximum value of $\mathtt{time}$. Add $(C, \mathcal{P}, t, y, \mathsf{com} \cdot \mathsf{com}_0, \mathtt{time}{++})$ to D.
  4. For each $i \in [n]$, if C is corrupt ask $\mathcal{A}$ for a bit b_i^*. If $b_i^* = 1$, compute $(\delta_i, \pi_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, f, i)$. Otherwise set $(\delta_i, \pi_i) := (\perp, \perp)$.
  5. If P_i is corrupt, ask $\mathcal{A}$ for values (δ_i', π_i') and set $(\delta_i, \pi_i) = (\delta_i', \pi_i')$. Output $(\mathtt{sid}, \mathsf{ZeroShare}, P_i, (C, \delta_i, \pi_i))$.
  6. Finally, output $(\mathtt{sid}, \mathsf{ZeroSetupDone}, C, (1))$.
- On input $(\mathtt{sid}, \mathsf{AuxRecover}, P_i, (C))$ for some client C, retrieve $(C, *, *, y, \mathsf{com}, \mathtt{time}) \in D$ for the maximum value of $\mathtt{time}$ and output $(\mathtt{sid}, \mathsf{AuxInfo}, P_i, (C, y, \mathsf{com}))$.

Figure 4: The encrypted secret sharing functionality $\mathcal{F}_{\mathsf{SS}}$.

– On input $(\mathtt{sid}, \mathsf{PKSetup}, P)$, it proceeds as follows:

1. Sample $\mathsf{dk} \xleftarrow{\$} \mathbb{Z}_p$ and set $\mathsf{ek} := g_2^{\mathsf{dk}}$.
2. Delete any existing entries $(P, *, *, *, *) \in L$ and add $(P, \mathsf{ek}, \mathsf{dk}, \mathtt{time} := 1, \mathtt{unleaked} := 1)$ to L.
3. Output $(\mathtt{sid}, \mathsf{PKSetupResult}, P, (\mathsf{ek}, \mathsf{dk}))$.

– On input $(\mathtt{sid}, \mathsf{PKRecover}, P, (Q))$, retrieve $(Q, \mathsf{ek}, *, *, *) \in L$ and output $(\mathtt{sid}, \mathsf{PKRecoverResult}, P, (Q, \mathsf{ek}))$.

Figure 5: The public key functionality $\mathcal{F}_{\mathsf{PK}}$.

To sign a message m on behalf of C, each institution (consisting of a hot and cold party) separately produces a partial signature on m. Upon receiving a signature request (which in most implementations would be passed to the cold party via the hot party), each component can honor the request by producing a partial signature σ_i^{hot} or σ_i^{cold}, respectively, which are combined by the hot party into σ_i.

Proving remembrance of each party's key material is done via CProof and HProof, respectively. We write that P_i^{cold} sends its proof directly to C, which in practice can be achieved by passing the message via P_i^{hot} assuming eventual delivery. Finally, the hot key shares can be proactively refreshed via an interactive protocol $\mathsf{ShareRefresh}$ between C and the hot parties, which is similar to $\mathsf{ClientRegister}$ and outputs some update information δ_i and a proof ζ_i of its correctness to each hot party.

Correctness Let $\mathcal{I}$ be the set of n institutions C registers with using threshold t. For $i \in [n]$, let ek_i be the output of $\mathsf{ColdRegister}$ for P_i^{cold}, vk be the output of $\mathsf{ClientRegister}$, com_T be the polynomial commitment after T executions of $\mathsf{ShareRefresh}$, and π_i^{hot} (resp. π_i^{cold}) be the output of HProof for P_i^{hot} and C (resp. CProof, P_i^{cold}, and C). For correctness, we require that for all T, if there exists some set $S = \{i : i \in [n]\}$ with $|S| \geq t$ such that $\Pi_{\mathsf{EKS}}.\mathsf{Vrfy}(\mathsf{crs}, \mathsf{com}_T, \pi_i^{\mathsf{hot}}) = 1$ and $\Pi_{\mathsf{DL}}.\mathsf{Vrfy}((\mathsf{ek}_i, g_2), \pi_i^{\mathsf{cold}}) = 1$ for all $i \in S$, then $\mathsf{BLS}.\mathsf{Vrfy}(\mathsf{vk}, m, \mathsf{BLS}.\mathsf{Reconstruct}(\{\sigma_i\}_{i \in S})) = 1$ with all but negligible probability, where σ_i is the output of TSign for P_i^{hot}, C, m.

Recall that TSign outputs computes σ_i as $\sigma_i^{\mathsf{hot}}/\sigma_i^{\mathsf{cold}}$, which by construction equals $H(m)^{\mathsf{sk}_i + \mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}})} \cdot H(m)^{-\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})} = H(m)^{\mathsf{sk}_i} = \mathsf{BLS}.\mathsf{TSign}(\mathsf{sk}_i, m)$. Thus correctness follows by construction, the soundness of $\Pi_{\mathsf{DL}}, \Pi_{\mathsf{EKS}}$, and the correctness of threshold BLS.

Efficiency and compactness of cold storage We wish to point out that our construction optimizes storage-, computation-, and communication-efficiency for

Parameters: Groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p with generators g_1, g_2, respectively; a degree-d KZG common reference string crs; hash functions $H : \{0,1\}^* \to \mathbb{G}_1$ and $\mathcal{H} : \mathbb{G}_2^2 \to \mathbb{Z}_p$ as defined in 2.4 and 5.5, resp.

Setup

Registering a cold storage P_i^{cold}: On input $(\mathtt{sid}, \mathsf{ColdRegister}, P_i^{\mathsf{cold}})$, P_i^{cold} calls $\mathcal{F}_{\mathsf{PK}}$ on input $(\mathtt{sid}, \mathsf{PKSetup}, P_i^{\mathsf{cold}})$ and receives response $(\mathtt{sid}, \mathsf{PKSetupResult}, P_i^{\mathsf{cold}}, (\mathsf{ek}_i, \mathsf{dk}_i))$. It stores $\mathsf{ek}_i, \mathsf{dk}_i$ and outputs $(\mathtt{sid}, \mathsf{ColdRegistered}, P_i^{\mathsf{cold}}, \mathsf{ek}_i)$.

Registering a client C: On input $(\mathtt{sid}, \mathsf{ClientRegister}, C, (t, \mathcal{I}))$, where $\mathcal{I} = \{(P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})\}_{i\in[n]}$ is a set of institutional entities and $t \le n$ a signing threshold:

1. C calls $\mathcal{F}_{\mathsf{PK}}$ on input $(\mathtt{sid}, \mathsf{PKRecover}, C, (P_i^{\mathsf{cold}}))$ for all $i \in [n]$. It waits until it receives a response $(\mathtt{sid}, \mathsf{PKRecoverResult}, C, (P_i^{\mathsf{cold}}, \mathsf{ek}_i))$ for all $i \in [n]$.

2. C calls $\mathcal{F}_{\mathsf{SS}}$ on input $(\mathtt{sid}, \mathsf{SSSetup}, C, (t, \{P_i^{\mathsf{hot}}\}_{i\in[n]}, \{\mathsf{ek}_i\}_{i\in[n]}))$. The latter outputs $(\mathtt{sid}, \mathsf{SecretShare}, P_i^{\mathsf{hot}}, (C, i, \tilde{x}_i, \pi_i))$ to P_i^{hot} $\forall i \in [n]$. It also outputs $(\mathtt{sid}, \mathsf{SSSetupDone}, C, (\mathsf{vk}, \{\mathsf{vk}_i\}_{i\in[n]}))$ to C.

3. Each P_i^{hot} stores the tuple $(C, \tilde{x}_i, \pi_i)$ in a list L_i. Then it outputs $(\mathtt{sid}, \mathsf{ClientRegistered}, P_i^{\mathsf{hot}}, (C, b_i = 1))$.

4. Meanwhile, C stores the parameters and the values it received from $\mathcal{F}_{\mathsf{SS}}$ in the tuple $D = (\mathsf{vk}, \{\mathsf{vk}_i\}_{i\in[n]}, t, \mathcal{I})$. Finally, it outputs $(\mathtt{sid}, \mathsf{ClientRegistered}, C, \mathsf{vk}, \{\mathsf{vk}_i\}_{i\in[n]})$.

Signing

Threshold signing: On input a signature request $(\mathtt{sid}, \mathsf{TSign}, P_i^{\mathsf{hot}}, (C, m))$ from a client C,

1. If P_i^{hot} decides to honor the request, it calls $\mathcal{F}_{\mathsf{SS}}$ on $(\mathtt{sid}, \mathsf{AuxRecover}, P_i^{\mathsf{hot}}, (C))$ to get vk and sends (vk, m) to P_i^{cold}. Otherwise, it aborts.

2. If P_i^{cold} decides to honor the request, it sends $\sigma_i^{\mathsf{cold}} := H(m)^r$ to P_i^{hot}, where $r := \mathcal{H}(\mathsf{vk}^{\mathsf{dk}_{i,1}}, \mathsf{vk}^{\mathsf{dk}_{i,2}})$.

3. Meanwhile, P_i^{hot} retrieves $(C, \tilde{x}_i, *) \in L_i$ and computes $\sigma_i^{\mathsf{hot}} := H(m)^{\tilde{x}_i}$.

4. Once it receives σ_i^{cold}, P_i^{hot} outputs $(\mathtt{sid}, \mathsf{TSignResult}, P_i^{\mathsf{hot}}, (C, m, \sigma_i := \sigma_i^{\mathsf{hot}}/\sigma_i^{\mathsf{cold}}))$.

Figure 6: Our BLS hot-cold threshold wallet protocol (setup and threshold signing).

Parameters: Groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p with generators g_1, g_2, respectively; a degree-d KZG common reference string crs.

Proofs of Remembrance

Cold proof: A client C can verify that an institutional cold storage P_i^{cold} still retains its key material (namely dk_i). On input $(\mathtt{sid}, \mathsf{CProof}, C, (P_i^{\mathsf{cold}}))$, C sends a designated proof request message to P_i^{cold}. Then:

1. P_i^{cold} retrieves its stored $\mathsf{ek}_i = (\mathsf{ek}_{i,1}, \mathsf{ek}_{i,2}), \mathsf{dk}_i = (\mathsf{dk}_{i,1}, \mathsf{dk}_{i,2})$ and computes a non-replayable proof $\pi_{i,k}^{\mathsf{cold}} \leftarrow \Pi_{\mathsf{DL}}.\mathsf{Prove}(\mathsf{ek}_{i,k}; \mathsf{dk}_{i,k})$ for $k = 1, 2$. It sends $\pi_i^{\mathsf{cold}} := (\pi_{i,1}^{\mathsf{cold}}, \pi_{i,2}^{\mathsf{cold}})$ to C.

2. To verify, C calls $\mathcal{F}_{\mathsf{PK}}$ on input $(\mathtt{sid}, \mathsf{PKRecover}, C, (P_i^{\mathsf{cold}}))$ and receives $(\mathtt{sid}, \mathsf{PKRecoverResult}, C, (P_i^{\mathsf{cold}}, \mathsf{ek}_i))$. It parses $\mathsf{ek}_i = (\mathsf{ek}_{i,1}, \mathsf{ek}_{i,2})$ and computes $b_k \leftarrow \Pi_{\mathsf{DL}}.\mathsf{Vrfy}(\mathsf{ek}_{i,k}, \pi_{i,k}^{\mathsf{cold}})$ for $k = 1, 2$. Finally it outputs $(\mathtt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b_1 \wedge b_2))$.

Hot proof: A client C can also verify that an institutional hot storage P_i^{cold} still retains its key material (namely $\tilde{x}_i$). On input $(\mathtt{sid}, \mathsf{HProof}, C, (P_i^{\mathsf{hot}}))$, C sends a designated proof request message to P_i^{hot}. Then:

1. P_i^{hot} calls $\mathcal{F}_{\mathsf{SS}}$ on input $(\mathtt{sid}, \mathsf{AuxRecover}, P_i^{\mathsf{hot}}, (C))$ and receives $(\mathtt{sid}, \mathsf{AuxInfo}, P_i^{\mathsf{hot}}, (C, \mathsf{vk}, \mathsf{com}))$. Then it uses com and the stored tuple $(C, \tilde{x}_i, \pi_i) \in L_i$ to compute $\pi_i^{\mathsf{hot}} \leftarrow \Pi_{\mathsf{EKS}}.\mathsf{Prove}((\mathsf{crs}, \mathsf{com}, i); (\tilde{x}_i, \pi_i))$, which it sends to C.

2. To verify, C calls $\mathcal{F}_{\mathsf{SS}}$ on input $(\mathtt{sid}, \mathsf{AuxRecover}, C, (C))$ and receives $(\mathtt{sid}, \mathsf{AuxInfo}, C, (C, *, \mathsf{com}))$. Then it lets $b \leftarrow \Pi_{\mathsf{EKS}}.\mathsf{Vrfy}((\mathsf{crs}, \mathsf{com}, i), \pi_i^{\mathsf{hot}})$ and outputs $(\mathtt{sid}, \mathsf{HProofResult}, C, (P_i^{\mathsf{hot}}, b))$.

Proactive Refresh

Hot share refresh: A client C can trigger a refresh of its hot key shares. On input $(\mathtt{sid}, \mathsf{ShareRefresh}, C)$, C retrieves its stored wallet parameters $D = (\mathsf{vk}, *, t, \mathcal{I})$ where $\mathcal{I} = \{(P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})\}_{i \in [n]}$. Then:

1. C calls $\mathcal{F}_{\mathsf{SS}}$ on intput $(\mathtt{sid}, \mathsf{ZeroSetup}, C, (t, \{P_i^{\mathsf{hot}}\}_{i \in [n]}))$.

2. $\mathcal{F}_{\mathsf{SS}}$ outputs $(\mathtt{sid}, \mathsf{ZeroShare}, P_i^{\mathsf{hot}}, (C, \delta_i, \zeta_i))$ to P_i^{hot} for all $i \in [n]$. It also outputs $(\mathtt{sid}, \mathsf{ZeroSetupDone}, C, (b))$ to C.

3. Each P_i^{hot} checks if $(\delta_i, \zeta_i) = (\perp, \perp)$. If so, it sets $b_i = 0$; otherwise, it sets $b_i = 1$ and updates $(C, \tilde{x}_i, \pi_i) \in L_i$ to $(C, \tilde{x}_i + \delta_i, \pi_i \cdot \zeta_i)$. Then it outputs $(\mathtt{sid}, \mathsf{ShareRefreshResult}, P_i^{\mathsf{hot}}, (C, b_i))$.

4. Meanwhile, C outputs $(\mathtt{sid}, \mathsf{ShareRefreshResult}, C, (b))$.

Figure 7: Our BLS hot-cold threshold wallet protocol (proofs of remembrance and proactive refresh).

the cold storage. Each P_i^{cold} only stores a single decryption key $\mathsf{dk}_i \in \mathbb{G}_2^2$, regardless of the number of clients registered with its institution. To produce a cold partial signature, it computes two $\mathbb{G}_2$ exponentiations, one addition in $\mathbb{Z}_p$, and a single evaluation of $\mathcal{H}$, which is highly efficient since it is a simple subset sum (see Section 5.5). Computing a proof of remembrance requires 2 $\mathbb{G}_2$ exponentiations, 1 hash function evaluation (for Fiat-Shamir), and 2 additions and multiplications each in $\mathbb{Z}_p$. Finally, in terms of communication, the cold storage only needs to send a single $\mathbb{G}_1$ element per signing operation. A cold proof of remembrance consists of 2 $\mathbb{G}_2$ and 2 $\mathbb{Z}_p$ elements.

5.7.1 Security Analysis

We prove our scheme secure in the universal composability framework[7], which we summarize in Section 2.8.

Messages to and from the ideal functionalities consist of two pieces: a *header* and the *contents*. The header normally consists of a session identifier `sid`, a description of the action the functionality should take/has taken, and the sender/recipient. In this paper, we will use the convention of putting the message contents in parentheses so it is clear where the (public) header ends and the (private) contents begin, e.g., $(\mathtt{sid}, \mathsf{Action}, P_{\mathsf{sender}}, (\mathsf{contents}))$ or $(\mathtt{sid}, \mathsf{ActionDone}, P_{\mathsf{recipient}}, \mathsf{publicvars}, (\mathsf{contents}))$.

Ideal Functionality We now present our hot-cold threshold BLS functionality $\mathcal{F}_{\mathsf{HC}}$, whose interfaces can only be called by *either* an institutional hot or cold storage (specified by the superscripts hot and cold) or a client. For readability, we split $\mathcal{F}_{\mathsf{HC}}$ into three figures. The first, Figure 8, describes how parties register in the system. The signing, share refresh, and proof of remembrance interfaces of $\mathcal{F}_{\mathsf{HC}}$ are described in Figure 9.

The most complicated part of the functionality is the signing interface, which provides partial BLS signatures σ_i on requested messages. For uncorrupted institutions I_i (that is, neither P_i^{cold} and P_i^{hot} are corrupt), σ_i is computed honestly. If the hot party has been corrupted but the cold remains honest, the functionality asks the adversary whether to use the correct value for the hot signature; if so, it computes σ_i correctly and also sends the cold signature to the adversary. Otherwise, it outputs $\sigma_i = \perp$ and sends $H(m)^r$ for a uniform r to $\mathcal{A}$ (as the "corresponding" cold signature). (The idea is that in this case, the hot signature is incorrect so σ_i will not verify, and the cold signature should be "useless" without it: it should look random to the adversary.) If the hot party is honest and the cold is corrupt, the functionality behaves in the same way but with the hot and cold roles reversed. Finally, if both parties in a pair are corrupt, the adversary will get no information about the hot or cold partial signatures from the functionality, which only outputs either the correct σ_i or $\perp$, depending on whether the adversary says to compute the hot and cold partial signatures correctly.

[7]Specifically, we use the version presented in [CLOS02].
[8]We assume share refreshes are sequential and delivery to all P_i precedes any new refreshes.

Public parameters: Groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p with generators g_1, g_2, respectively; a degree-d KZG common reference string crs; a hash function $H : \{0,1\}^* \to \mathbb{G}_1$; an extractor $\mathcal{H} : \mathbb{G}_2^2 \to \mathbb{Z}_p$.
Internal variables:
$L_{\mathsf{cold}} = \{(P^{\mathsf{cold}}, \mathsf{ek}, \mathsf{dk}, \mathtt{leaked}, \mathtt{tampered}, \mathtt{corrupted}, \mathtt{allowc})\}$, a table with the state of each cold storage's key material.
$D = \{(C, \mathcal{I}, t, y, \mathsf{com})\}$, a table of registered clients and their metadata.
$L_{\mathsf{hot}} = \{(P^{\mathsf{hot}}, C_j, \tilde{x}, \pi, \mathtt{time}, \mathtt{leaked}, \mathtt{tampered})\}$, which keeps track of the key material of every hot storage for each of its clients, and whether it has been leaked or tampered with.
$S_{\mathsf{hot}} = \{P^{\mathsf{hot}}, \mathtt{corrupted}, \mathtt{allowc}\}$, which keeps track of whether each hot party has been corrupted.

Registration

- On input $(\mathtt{sid}, \mathsf{ColdRegister}, P_i^{\mathsf{cold}})$, it proceeds as follows:
  1. Sample $\mathsf{dk}_i \xleftarrow{\$} \mathbb{Z}_p$ and set $\mathsf{ek}_i := g_2^{\mathsf{dk}_i}$.
  2. Delete any existing entries for P_i^{cold} in L_{cold} and add $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathsf{dk}_i, \mathtt{leaked} := 0, \mathtt{tampered} := 0, \mathtt{corrupted} := 0, \mathtt{allowc} := 1)$ to L_{cold}.
  3. Output $(\mathtt{sid}, \mathsf{ColdRegistered}, P_i^{\mathsf{cold}}, \mathsf{ek}_i)$.
- On input $(\mathtt{sid}, \mathsf{ClientRegister}, C, (t, \mathcal{I}))$, where $\mathcal{I} = \{(P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})\}_{i \in [n]}$ is a set of institutional entities and $t \leq n$ a signing threshold, it proceeds as follows:
  1. For each $i \in [n]$, retrieve $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, *, *, *, *, *)$ from L_{cold}. If a public key for some party is unavailable then output $y := \bot$.
  2. Otherwise, sample $x \xleftarrow{\$} \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
  3. Generate t-out-of-n Shamir Shares of x as $x_1, \dots x_n \in \mathbb{Z}_p$. Let $y_i := g_2^{x_i}$ $\forall i \in [n]$.
  4. Interpolate the degree-n polynomial $\tilde{f}$ such that $\tilde{f}(i) = \mathcal{H}(\mathsf{ek}_i^x) + x_i$ $\forall i \in [n]$. Compute $\mathsf{com} \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, \tilde{f})$.
  5. Delete any existing entries $(C, *, *, *, *) \in D$ and add $(C, \mathcal{I}, t, y, \mathsf{com})$ to D. Send $(\mathtt{sid}, \mathsf{ClientRegistered}, C, y, \{y_i\}_{i \in [n]})$ to C.
  6. For each $i \in [n]$:
     (a) Compute $(\tilde{x}_i, \pi_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, \tilde{f}, i)$.
     (b) Output $(\mathtt{sid}, \mathsf{ClientRegistered}, P_i^{\mathsf{hot}}, (C, b_i = 1))$.
     (c) Once $\mathcal{A}$ has delivered/allowed delivery of the message, delete any existing entries $(P_i^{\mathsf{hot}}, C, *, *, *, *, *) \in L_{\mathsf{hot}}$ and add $(P_i^{\mathsf{hot}}, C, \tilde{x}_i, \pi_i, \mathtt{time} := 0, \mathtt{leaked} := 0, \mathtt{tampered} := 0)$ to L_{hot}. Delete any existing entries $(P_i^{\mathsf{hot}}, \mathtt{corrupted}, \mathtt{allowc}) \in S_{\mathsf{hot}}$ and add $(P_i^{\mathsf{hot}}, \mathtt{corrupted} := 0, \mathtt{allowc} := 1)$.

Figure 8: The BLS hot-cold threshold wallet functionality $\mathcal{F}_{\mathsf{HC}}$ (registration).

Signing

– On input $(\texttt{sid}, \mathsf{TSign}, P_i^{\mathsf{hot}}, (C, m))$, retrieve $(C, \mathcal{I}, *, \mathsf{vk}, *) \in D$ and $(P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}}) \in \mathcal{I}$. Then:

1. Get $(P_i^{\mathsf{cold}}, *, \mathsf{dk}_i, *, \texttt{tampered}, \texttt{corrupt}_{\mathsf{cold}}, \texttt{allowc}) \in L_{\mathsf{cold}}$. If $\texttt{allowc} = 1$, set it to 0.
   - If $\texttt{corrupt}_{\mathsf{cold}} = 1$, send $(\texttt{sid}, \mathsf{CSignRequest}, \mathcal{A}, (P_i^{\mathsf{cold}}, C, m))$ to $\mathcal{A}$, wait for response b, and set $b_{\mathsf{cold}} := (b \wedge \neg\texttt{tampered})$. Otherwise $(\texttt{corrupt}_{\mathsf{cold}} = 0)$, set $b_{\mathsf{cold}} := \neg\texttt{tampered}$.
2. Retrieve $(P_i^{\mathsf{hot}}, C, \tilde{x}_i, *, \texttt{time}, *, \texttt{tampered}, \texttt{corrupt}_{\mathsf{hot}}, *) \in L_{\mathsf{hot}}$ for the maximum $\texttt{time}$.
   - If $\texttt{corrupt}_{\mathsf{hot}} = 1$, send $(\texttt{sid}, \mathsf{HSignRequest}, \mathcal{A}, (P_i^{\mathsf{hot}}, C, m))$ to $\mathcal{A}$, wait for response b, and set $b_{\mathsf{hot}} := (b \wedge \neg\texttt{tampered})$. Otherwise, set $b_{\mathsf{hot}} := \neg\texttt{tampered}$.
3. If $b_{\mathsf{cold}} \wedge b_{\mathsf{hot}}$, compute $\sigma_i^{\mathsf{cold}} := H(m)^{\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})}$ and $\sigma_i^{\mathsf{hot}} := H(m)^{\tilde{x}_i}$. Let $\sigma_i := \sigma_i^{\mathsf{hot}} / \sigma_i^{\mathsf{cold}}$. Output $(\texttt{sid}, \mathsf{TSignResult}, P_i^{\mathsf{hot}}, (C, m, \sigma_i))$.
   - If $(\texttt{corrupt}_{\mathsf{cold}} \wedge \neg\texttt{corrupt}_{\mathsf{hot}})$, also send σ_i^{hot} to $\mathcal{A}$. If $(\neg\texttt{corrupt}_{\mathsf{cold}} \wedge \texttt{corrupt}_{\mathsf{hot}})$, send σ_i^{cold} to $\mathcal{A}$.
   - Additionally, for every party P_j^{hot} such that $(P_j^{\mathsf{hot}}, *) \in \mathcal{I}$, retrieve $(P_j^{\mathsf{hot}}, *, \texttt{allowc}) \in S_{\mathsf{hot}}$ and set $\texttt{allowc}$ to 0.
4. If instead $\neg(b_{\mathsf{cold}} \wedge b_{\mathsf{hot}})$, output $(\texttt{sid}, \mathsf{TSignResult}, P_i^{\mathsf{hot}}, (C, m, \perp))$.
   - If $(\texttt{corrupt}_{\mathsf{cold}} \wedge \neg\texttt{corrupt}_{\mathsf{hot}})$ or $(\neg\texttt{corrupt}_{\mathsf{cold}} \wedge \texttt{corrupt}_{\mathsf{hot}})$, also sample $r \xleftarrow{\$} \mathbb{Z}_p$ and send $H(m)^r$ to $\mathcal{A}$.

Proofs of Remembrance

– On input $(\texttt{sid}, \mathsf{CProof}, P_i^{\mathsf{cold}}, (C))$, retrieve $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathsf{dk}_i, *, *, \texttt{corrupted}, *) \in L_{\mathsf{cold}}$. If $\texttt{corrupted} = 1$, send $(\texttt{sid}, \mathsf{CProofRequest}, \mathcal{A}, (P_i^{\mathsf{cold}}))$ to the adversary, who will send back a bit b^* to represent whether the query should be responded to honestly. Set $b := b^* \wedge (\mathsf{ek}_i = g_2^{\mathsf{dk}_i})$ and output $(\texttt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$.

– On input $(\texttt{sid}, \mathsf{HProof}, C, (P_i^{\mathsf{hot}}))$, retrieve $(P_i^{\mathsf{hot}}, C, \tilde{x}_i, \pi_i, *, *, *, \texttt{corrupted}, *) \in L_{\mathsf{hot}}$ and $(C, *, *, *, \mathsf{com}) \in D$. If $\texttt{corrupted} = 1$, send $(\texttt{sid}, \mathsf{HProofRequest}, \mathcal{A}, (P_i^{\mathsf{hot}}))$ to the adversary, who will send back a bit b^*. Set $b := b^* \wedge \mathsf{KZG.Vrfy}(\mathsf{crs}, \mathsf{com}, i, \tilde{x}_i, \pi_i)$ and output $(\texttt{sid}, \mathsf{HProofResult}, C, (P_i^{\mathsf{hot}}, b))$.

Proactive Refresh

– On input $(\texttt{sid}, \mathsf{ShareRefresh}, C)$, it proceeds as follows:

1. Output $\perp$ if a prior share refresh from C is still pending.[8]
2. Retrieve $(C, \mathcal{I}, t, y, \mathsf{com}, \texttt{time}) \in D$ for the maximum value of $\texttt{time}$.
3. Generate t-out-of-n Shamir shares of 0 as $x_1, \ldots, x_n \in \mathbb{Z}_p$ using polynomial f

We will argue that this implements threshold BLS signatures, so the security of a protocol implementing $\mathcal{F}_{\mathsf{HC}}$ follows from security of threshold BLS.

Adversarial interference Figure 10 gives the leak and tamper interfaces of $\mathcal{F}_{\mathsf{HC}}$, which an adversary can use to interfere with the information stored by the hot and cold storages in the system. We also give an explicit corruption interface, which only allows non-adaptive corruptions of the cold and hot parties (in the latter case, on a per-epoch basis). The interfaces also capture the fact that the client C is out of scope for corruption.

Theorem 2 (security)**.** *Our hot-cold threshold wallet construction (??) UC-realizes $\mathcal{F}_{\mathsf{HC}}$ in the $\mathcal{F}_{\mathsf{SS}}, \mathcal{F}_{\mathsf{PK}}$-hybrid model.*

Proof. Let $\mathcal{A}$ be the adversary interacting with the parties (namely, $P_1^{\mathsf{hot}}, P_1^{\mathsf{cold}}, \ldots, P_n^{\mathsf{hot}}, P_n^{\mathsf{cold}}, C$) running the protocol presented in ??. We will construct a simulator $\mathcal{S}$ running in the ideal world against $\mathcal{F}_{\mathsf{HC}}$ so that no environment $\mathcal{E}$ can distinguish an execution of the ideal-world interaction from the real protocol. $\mathcal{S}$ will interact with $\mathcal{F}_{\mathsf{HC}}$, $\mathcal{E}$, and invoke a copy of $\mathcal{A}$ to run a simulated interaction of the protocol (we call this simulated interaction between $\mathcal{A}$, $\mathcal{E}$, and the parties the *internal interaction* to distinguish it from the *external interaction* between $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{F}_{\mathsf{HC}}$).

Each protocol/algorithm begins with a party receiving some input. In the ideal world, this input is received by some dummy party, who immediately copies it to its outgoing communication tape where $\mathcal{S}$ can read it (public header only) and potentially deliver it to the ideal functionality $\mathcal{F}_{\mathsf{HC}}$. (Recall that for simplicity we assume authenticated communication, so $\mathcal{S}$ cannot modify the messages it delivers to/from $\mathcal{F}_{\mathsf{HC}}$.) To complete the protocol, $\mathcal{S}$ will deliver the response of $\mathcal{F}_{\mathsf{HC}}$ to the dummy party, who copies it to its output tape (which is visible to $\mathcal{E}$).

Message delivery $\mathcal{S}$ waits to deliver any messages until $\mathcal{A}$ delivers the corresponding message in the internal interaction.

Corruptions Whenever $\mathcal{A}$ corrupts a party P_i^{cold} or P_i^{hot} in the internal execution, $\mathcal{S}$ corrupts the corresponding dummy party (via the $\mathsf{Corrupt}$ interface).

Leak and Tamper Whenever $\mathcal{A}$ leaks or tampers with the inputs of a party in the internal execution, $\mathcal{S}$ uses the corresponding interface of $\mathcal{F}_{\mathsf{HC}}$ to learn/change the same information (it can use any functions f, g for tampering).

Cold Registration $\mathcal{S}$ will deliver the message $(\mathtt{sid}, \mathsf{ColdRegister}, P_i^{\mathsf{cold}})$ from $\tilde{P}_i^{\mathsf{cold}}$ to $\mathcal{F}_{\mathsf{HC}}$ once $\mathcal{A}$ delivers $(\mathtt{sid}, \mathsf{PKSetup}, P_i^{\mathsf{cold}})$ to $\mathcal{F}_{\mathsf{PK}}$ in the internal interaction.

– If $\tilde{P}_i^{\mathsf{cold}}$ is corrupt, $\mathcal{S}$ records $(P_i^{\mathsf{cold}}, \mathsf{ek}_i^*, \mathtt{corrupt} = 1)$ in a local database L'_{cold}, where ek_i^* is the value sent by $\mathcal{A}$ on (the corrupted) $\tilde{P}_i^{\mathsf{cold}}$'s behalf.

Leak

- On input $(\texttt{sid}, \mathsf{Leak}, \mathcal{A}, (P_i^{\mathsf{hot}}))$, for every entry $(P_i^{\mathsf{hot}}, C_j, \tilde{x}_i, \pi_i, \texttt{time}, \texttt{leaked}, *) \in L_{\mathsf{hot}}$, set $\texttt{leaked}$ to 1 and send $(\texttt{sid}, \mathsf{LeakResult}, \mathcal{A}, ((P_i^{\mathsf{hot}}, C_j, \tilde{x}_i, \pi_i, \texttt{time}))$ to $\mathcal{A}$.
- On input $(\texttt{sid}, \mathsf{Leak}, \mathcal{A}, (P_i^{\mathsf{cold}}))$, retrieve the entry $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathsf{dk}_i, \texttt{leaked}, *, *, *) \in L_{\mathsf{cold}}$. Set $\texttt{leaked}$ to 1 and send $(\texttt{sid}, \mathsf{LeakResult}, \mathcal{A}, ((P_i^{\mathsf{cold}}, \mathsf{dk}_i))$ to $\mathcal{A}$.

Tamper

- On input $(\texttt{sid}, \mathsf{Tamper}, \mathcal{A}, (P_i^{\mathsf{hot}}, C, f, g))$, where f, g are functions:
  1. Retrieve $(P_i^{\mathsf{hot}}, C, \tilde{x}_i, \pi_i, \texttt{time}, \texttt{leaked}, \texttt{tampered}) \in L_{\mathsf{hot}}$ for the maximum value of $\texttt{time}$. Let $\tilde{x}_i' := f(\tilde{x}_i)$ and $\pi_i' := g(\pi_i)$.
  2. If $\tilde{x}_i', \pi_i' \neq \perp$, let $b := 1$ and update the entry to $(C, P_i^{\mathsf{hot}}, \tilde{x}_i', \pi_i', \texttt{time}, \texttt{leaked}, \texttt{tampered} = 1)$. Otherwise let $b := 0$.
  3. Send $(\texttt{sid}, \mathsf{TamperDone}, \mathcal{A}, (C, P_i^{\mathsf{hot}}, b))$ to $\mathcal{A}$.
- On input $(\texttt{sid}, \mathsf{Tamper}, \mathcal{A}, (P_i^{\mathsf{cold}}, f))$, where f is a function:
  1. Retrieve $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathsf{dk}_i, \texttt{leaked}, \texttt{tampered}, \texttt{corrupt}, \texttt{allowc}) \in L_{\mathsf{cold}}$. Let $\mathsf{dk}_i' := f(\mathsf{dk}_i)$.
  2. If $\mathsf{dk}_i' \neq \perp$, let $b := 1$ and update the entry to $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathsf{dk}_i', \texttt{leaked}, \texttt{tampered} = 1, \texttt{corrupt}, \texttt{allowc})$. Otherwise let $b := 0$.
  3. Send $(\texttt{sid}, \mathsf{TamperDone}, \mathcal{A}, (P_i^{\mathsf{cold}}, b))$ to $\mathcal{A}$.

Corrupt

- On input $(\texttt{sid}, \mathsf{Corrupt}, \mathcal{A}, (P_i^{\mathsf{hot}}))$, retrieve $(P_i^{\mathsf{hot}}, \texttt{corrupted}, \texttt{allowc}) \in S_{\mathsf{hot}}$ and check that $\texttt{allowc} = 1$. If so, set $\texttt{corrupted}$ to 1. ($\mathcal{A}$ receives P_i^{hot}'s tapes and $\mathcal{E}$ is notified.)
- On input $(\texttt{sid}, \mathsf{Corrupt}, \mathcal{A}, (P_i^{\mathsf{cold}}))$, retrieve $(P_i^{\mathsf{cold}}, *, *, *, *, \texttt{corrupted}, \texttt{allowc}) \in L_{\mathsf{cold}}$ and check if $\texttt{allowc} = 1$. If so, set $\texttt{corrupted}$ to 1. ($\mathcal{A}$ receives P_i^{cold}'s tapes and $\mathcal{E}$ is notified.)

Figure 10: The BLS hot-cold threshold wallet functionality $\mathcal{F}_{\mathsf{HC}}$ (adversarial interfaces).

– Otherwise, if $\tilde{P}_i^{\mathsf{cold}}$ is honest, it stores $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \texttt{corrupt} = 0) \in L'_{\mathsf{cold}}$, where ek_i is the value from $\mathcal{F}_{\mathsf{HC}}$'s response.

It delivers the response to $\tilde{P}_i^{\mathsf{cold}}$ once $\mathcal{A}$ delivers the result of $\mathcal{F}_{\mathsf{PK}}$ in the internal interaction.

Client Registration and Share Refreshes Recall that C is out of scope for corruptions. Thus, in any case $\mathcal{S}$ will deliver the message $(\texttt{sid}, \mathsf{ClientRegister}, C, (t, \mathcal{I}))$ on $\tilde{C}$'s outgoing communication tape to $\mathcal{F}_{\mathsf{HC}}$ once $\mathcal{A}$ delivers $(\texttt{sid}, \mathsf{SSSetup}, C, (t, \{P_i^{\mathsf{hot}}\}_{i\in[n]}, *)$ from C to $\mathcal{F}_{\mathsf{SS}}$ in the internal interaction. $\mathcal{F}_{\mathsf{HC}}$ will send $(\texttt{sid}, \mathsf{ClientRegistered}, C, (\mathsf{vk}))$ to $\tilde{C}$ and $(\texttt{sid}, \mathsf{ClientRegistered}, P_i^{\mathsf{hot}}, (C, b_i))$ to $\tilde{P}_i^{\mathsf{hot}}$. $\mathcal{S}$ delivers these messages to $\tilde{C}$ and each *honest* party $\tilde{P}_i^{\mathsf{hot}}$, respectively, once the corresponding response by $\mathcal{F}_{\mathsf{SS}}$ is delivered to them in the internal interaction. For any corrupt $\tilde{P}_i^{\mathsf{hot}}$, $\mathcal{S}$ instead outputs on its behalf the same bit b_i as $\mathcal{A}$ does in the internal interaction. The simulation for share refreshes works the same way.

Signing We will use the fact that partial BLS signatures σ_i can be verified with respect to the partial verification key vk_i. Let $\mathsf{PVrfy}(\mathsf{vk}_i, m, \sigma_i)$ be the partial verification algorithm, which in the case of BLS consists of checking that $(g_2, \mathsf{vk}_i, H(m), \sigma_i)$ is a co-Diffie-Hellman tuple (see **??**). On input $(\texttt{sid}, \mathsf{TSign}, P_i^{\mathsf{hot}}, (C, m))$, the simulator first delivers the message to $\mathcal{F}_{\mathsf{HC}}$ once $\mathcal{A}$ delivers the corresponding request from C to P_i^{hot} in the internal interaction. Then it retrieves the identity of the corresponding P_i^{cold} and behaves as follows:

– If $P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}}$ are both honest, $\mathcal{S}$ immediately delivers $\mathcal{F}_{\mathsf{HC}}$'s output $(\texttt{sid}, \mathsf{TSignResult}, P_i^{\mathsf{hot}}, (C, m, \sigma_i))$.

– If P_i^{hot} is honest and P_i^{cold} is corrupt, $\mathcal{S}$ will receive a $\mathsf{CSignRequest}$ from $\mathcal{F}_{\mathsf{HC}}$ to which it must respond with a bit b. It looks at the values σ_i^{hot} and $\sigma_i^{\mathsf{cold}*}$ in the internal interaction (the latter is output by $\mathcal{A}$ on behalf of the corrupt P_i^{cold}) and responds with $b := \mathsf{PVrfy}(\mathsf{vk}_i, m, \sigma_i^{\mathsf{hot}}/\sigma_i^{\mathsf{cold}*})$, where vk_i, m are the ith partial verification key and message requested in the *internal* execution (all known to $\mathcal{S}$). As before, it delivers $\mathcal{F}_{\mathsf{HC}}$'s output to $\tilde{C}$ immediately.

– If P_i^{hot} is corrupt and P_i^{cold} is honest, $\mathcal{S}$ will receive an $\mathsf{HSignRequest}$ from $\mathcal{F}_{\mathsf{HC}}$ to which it must again respond with a bit b. Similarly, it now looks at the values $\sigma_i^{\mathsf{hot}*}$ (output by $\mathcal{A}$) and σ_i^{cold} in the internal execution and responds with $b := \mathsf{PVrfy}(\mathsf{vk}_i, m, \sigma_i^{\mathsf{hot}*}/\sigma_i^{\mathsf{cold}})$. Again it immediately delivers $\mathcal{F}_{\mathsf{HC}}$'s output to $\tilde{C}$.

– Finally, if both P_i^{hot} and P_i^{cold} are corrupt, $\mathcal{S}$ checks if $\mathsf{PVrfy}(\mathsf{vk}_i, m, \sigma_i^{\mathsf{hot}*}/\sigma_i^{\mathsf{cold}*}) = 1$ in the internal execution. If so, it responds 1 to both $\mathsf{CSignRequest}$ and $\mathsf{HSignRequest}$; otherwise (w.l.o.g.) it sends 0 to both. Then it delivers $\mathcal{F}_{\mathsf{HC}}$'s output to $\tilde{C}$.

Proofs of Remembrance On input $(\texttt{sid}, \mathsf{CProof}, C, (P_i^{\mathsf{cold}}))$, the simulator delivers the message to $\mathcal{F}_{\mathsf{HC}}$ once $\mathcal{A}$ delivers the corresponding request from C to P_i^{cold} in the internal interaction.

– If $\tilde{P}_i^{\mathsf{cold}}$ is honest, $\mathcal{S}$ delivers $(\texttt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$ to $\tilde{C}$ once $\mathcal{A}$ delivers the output of $\mathcal{F}_{\mathsf{PK}}$ to C in the internal execution.

– On the other hand, if $\tilde{P}_i^{\mathsf{cold}}$ is corrupted, $\mathcal{S}$ will receive a message from $\mathcal{F}_{\mathsf{HC}}$ requesting a bit b^*. It retrieves the party's ek_i from L'_{cold} and the proof π_i^{cold} computed by $\mathcal{A}$ in the internal execution and sends back $b^* := \Pi_{\mathsf{DL}}.\mathsf{Vrfy}((\mathsf{ek}_i, g_2), \pi_i^{\mathsf{cold}})$. Finally it delivers $\mathcal{F}_{\mathsf{HC}}$'s output $(\texttt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$ to $\tilde{C}$ once $\mathcal{A}$ delivers the message from $\mathcal{F}_{\mathsf{PK}}$ to C in the internal execution.

On input $(\texttt{sid}, \mathsf{HProof}, P_i^{\mathsf{hot}}, (C))$, $\mathcal{S}$ acts in the same way as CProof, except in the corrupted case it sets $b^* := \Pi_{\mathsf{EKS}}.\mathsf{Vrfy}(\mathsf{com}, \pi_i^{\mathsf{hot}})$, where both $\mathsf{com}, \pi_i^{\mathsf{cold}}$ are the party's values in the internal execution.

It is straightforward to see that the simulated registration and share refreshes are identical to a real-world execution. The simulation of the signing protocol is computationally indistinguishable from the real-world execution by the correctness and security of threshold BLS signatures, which guarantees that $\mathcal{S}$ sends $b = 1$ iff $\sigma_i^{\mathsf{cold}*} = H(m)^{\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})}$, resp. $\sigma_i^{\mathsf{hot}*} = H(m)^{\tilde{x}_i}$. Similarly, the indistinguishability of simulating proofs of remembrance follows from the correctness and soundness of Π_{DL} and Π_{EKS}. $\square$

Finally, notice that the signing interface of $\mathcal{F}_{\mathsf{HC}}$ is identical to the functionality offered by threshold BLS, except that $\mathcal{A}$ additionally learns either σ_i^{hot} or σ_i^{cold} (in step 3) or a uniform value $H(m)^r$ (in step 4). Clearly the uniform value is independent of any private information can be simulated perfectly as a uniform $\mathbb{G}_2$ element. As for step 3, the simulator can again send a uniform $\mathbb{G}_2$ element, now in place of σ_i^{hot} (alternatively, σ_i^{cold}), and the simulation is statistically indistinguishable by Lemma 1.

5.7.2 Implementation and Evaluation

We implemented our construction in Rust using the BLS12-381 elliptic curve.[9] Each element in $\mathbb{G}_1$ is 48 bytes in compressed form, $\mathbb{G}_2$ is 96 bytes, and $\mathbb{Z}_p$ is 32 bytes. Thus the sizes of vk and ek_i are both 96 bytes. The cold and hot shares dk_i and $\tilde{x}_i$, as well as the share refresh information δ_i, are all 32 bytes. The (partial) signatures $\sigma_i^{\mathsf{hot}}, \sigma_i^{\mathsf{cold}}$, and σ_i, as well as the commitment com_T, are 48 bytes.

We report the runtimes of each of our algorithms for small $(t, n) = (3, 5)$, medium $(5, 20)$, and large $(67, 100)$ parameter settings. The times for cold registration and proofs, threshold signing, and processing hot share refreshes (which includes running $\Pi_{\mathsf{Ref}}.\mathsf{HVrfy}$ to check ζ_i) are all independent of (t, n) and are shown in Table 3. Hot share generation ($\mathsf{ClientRegister}$), producing hot

[9] https://github.com/hyperledger-labs/agora-key-share-proofs/

share refreshes (in our implementation, by C running the code of $\mathcal{F}_{\mathsf{SS}}$),Noemi: does this include $\Pi_{\mathsf{Ref}}.\mathsf{UCVrfy}$? i.e., checking that the update is a sharing of zero (verify KZG opening at zero) and that it's degree-t (check dcom). So, it's the bullet of $\mathsf{ClientUpdate}$ in spec.tex that starts with "The system should check that..." and hot proofs depend on the specific values of (t, n) and are shown in Table 4. The benchmarks are an average over 1000 runs using a machine with 8-core AMD Ryzen 9 5900HX with 64GB RAM and cache: L1 512KB, L2 4MB, L3 16 MB at 5GHz. The proof sizes are given in Table 5.

	Time
$\mathsf{ColdRegister}$	360μs
TSign	890μs
$\mathsf{ShareRefresh}$ (P_i^{hot})	5ms
Cold Prove ($\Pi_{\mathsf{DL}}.\mathsf{Prove}$)	370μs
Cold Verify ($\Pi_{\mathsf{DL}}.\mathsf{Vrfy}$)	560μs

Table 3: Benchmarks of our algorithms regardless of threshold

	Time (ms)		
$(t, n) =$	$(3, 5)$	$(5, 20)$	$(67, 100)$
$\mathsf{ClientRegister}$	10	40	170
$\mathsf{ShareRefresh}$ (C)	11	41	172
Hot Prove ($\Pi_{\mathsf{EKS}}.\mathsf{Prove}$)	10	40	170
Hot Verify ($\Pi_{\mathsf{EKS}}.\mathsf{Vrfy}$)	14	43	4

Table 4: Benchmarks of our algorithms for each setting of (t, n). Times for CProof and HProof are not included since they are identical to the corresponding rows of Table 5. The $\mathsf{ShareRefresh}$ times for C and P_i^{hot} include the UCVrfy resp. HVrfy times of the refresh proofs in Table 5.

	$\lvert\pi\rvert$ (B)
Cold proof (Π_{DL})	256
Hot proof (Π_{EKS})	304
Refresh proof (ζ_i)	48

Table 5: Sizes for our various proof systems regardless of threshold

5.8 Extensions

5.8.1 Trustless proactive refresh using a bulletin-board

Recall from Section 5.6.2 that the correctness of our proactive refresh protocols relies on the update polynomial $z_T(X)$ being of degree $t-1 \le d$ and having $z_T(0) = 0$. We can use a folklore technique [CHM+20, §2.5] to enforce the degree requirement: the client publishes an additional public "degree commitment" $\mathsf{dcom}_T := g_1^{\tau^{d-t+1} \cdot z_T(\tau)}$, which is verified by checking

$$e(\mathsf{dcom}_T, g_2) = e(\mathsf{ucom}_T, g_2^{\tau^{d-t+1}}),$$

where d is the degree of the KZG CRS. This ensures that the polynomial $z_T(X)$ committed to by ucom_T is of degree at most $t-1$. To enforce the evaluation at zero, C can simply provide a KZG opening proof for ucom_T at $X = 0$.

Additionally, the hot parties should also be sure that the client is using the *same* polynomial $z_T(X)$ for all of them when computing their share updates δ_i. This is easily done, since each P_i^{hot} is already provided with an evaluation proof ζ_i for δ_i. In the case of a trusted client, these were assumed to be computed correctly so P_i^{hot} only used them to blindly update the its key share and opening proof $(\tilde{x}_i, \pi_i)$. In the case of an untrusted client, parties will instead first check their correctness by ensuring $\mathsf{KZG.Vrfy}(\mathsf{crs}, \mathsf{ucom}_T, i, \delta_i, \zeta_i) = 1$, and only update their key share and opening proof if verification passes.

Figure 11 gives the proof system Π_{Ref} used to prove correctness of share refreshes, with verification split between verifying the well-formedness of the update polynomial $z_T(X)$ committed to by ucom_T (via UCVrfy) and of each hot party's refresh information δ_i (via HVrfy).

Given Π_{Ref}, it is fairly simple to realize $\mathcal{F}_{\mathsf{SS}}$ in a manner that avoids trusting C except for the setup phase. To do this, we assume a public bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$ with limited programmability (Figure 12). This functionality will store the most up-to-date commitment com to the hot shares $\tilde{x}_1, \ldots, \tilde{x}_n$. Whenever the shares are refreshed, it will also check the correctness of the update to com (namely ucom) before making the new commitment available.

Specifically, instead of only running the steps of $\mathcal{F}_{\mathsf{SS}}$ locally, C will compute some additional values using Π_{Ref} to let $\mathcal{F}_{\mathsf{BB}}$ and each P_i^{hot} check the correctness of the share and update commitments. The proof for the update commitment ucom will be checked by $\mathcal{F}_{\mathsf{BB}}$ before changing the stored commitment. Additionally, C will let $\mathcal{F}_{\mathsf{BB}}$ store and distribute ucom instead. We describe the full protocol in Figure 13, where changes with respect to locally running the ideal functionality $\mathcal{F}_{\mathsf{SS}}$ are shown in blue.

Theorem 3. *The encrypted secret sharing protocol in Figure 13 UC-realizes $\mathcal{F}_{\mathsf{SS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model.*

Proof. todo: □

6 Conclusion

References

[ADE$^+$20] Nabil Alkeilani Alkadri, Poulami Das, Andreas Erwig, Sebastian Faust, Juliane Krämer, Siavash Riahi, and Patrick Struck. Deterministic wallets in a quantum world. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1017–1031. ACM Press, November 2020. *[pages 21 and 12.]*

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *USENIX Security 2008*, pages 335–348. USENIX Association, July / August 2008. *[page 18.]*

[AGKK19] Myrto Arapinis, Andriana Gkaniatsou, Dimitris Karakostas, and Aggelos Kiayias. A formal treatment of hardware wallets. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 426–445. Springer, Heidelberg, February 2019. *[pages 20 and 12.]*

[AHS20] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. Cryptology ePrint Archive, Report 2020/1390, 2020. https://eprint.iacr.org/2020/1390. *[pages 22 and 14.]*

[AMPR19] Navid Alamati, Hart Montgomery, Sikhar Patranabis, and Arnab Roy. Minicrypt primitives with algebraic structure and applications. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 55–82. Springer, Heidelberg, May 2019. *[pages 10 and 9.]*

[ANO$^+$22] Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *2022 IEEE Symposium on Security and Privacy*, pages 2554–2572. IEEE Computer Society Press, May 2022. *[pages 22 and 14.]*

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046. *[page 17.]*

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete efficiency of probabilistically-checkable proofs. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 585–594. ACM Press, June 2013. *[page 17.]*

[BCK$^+$22] Mihir Bellare, Elizabeth C. Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than advertised security for non-interactive threshold signatures. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 517–550. Springer, Heidelberg, August 2022. *[pages 22 and 14.]*

[BEHG20] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Advances in Financial Technologies*, 2020. *[page 18.]*

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280. Springer, Heidelberg, April 2012. *[pages 18 and 19.]*

[BHK$^+$19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2129–2146. ACM Press, November 2019. *[page 18.]*

[BLM22] Michele Battagliola, Riccardo Longo, and Alessio Meneghetti. Extensible decentralized secret sharing and application to schnorr signatures. Cryptology ePrint Archive, Report 2022/1551, 2022. https://eprint.iacr.org/2022/1551. *[pages 22 and 14.]*

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, December 2001. *[pages 7, 22, and 14.]*

[BMP22] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. Efficient asymmetric threshold ECDSA for MPC-based cold storage. Cryptology ePrint Archive, Report 2022/1296, 2022. https://eprint.iacr.org/2022/1296. *[pages 21, 24, 12, and 16.]*

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003. *[pages 22 and 14.]*

[But14] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *White paper*, 2014. *[page 5.]*

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. *[page 10.]*

[CCL$^+$20] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 266–296. Springer, Heidelberg, May 2020. *[pages 22 and 14.]*

[CCL$^+$21] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA revisited: Online/offline extensions, identifiable aborts, proactivity and adaptive security. Cryptology ePrint Archive, Report 2021/291, 2021. https://eprint.iacr.org/2021/291. *[pages 22 and 14.]*

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Heidelberg, February / March 2013. *[page 20.]*

[CGG$^+$20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1769–1787. ACM Press, November 2020. *[pages 22 and 14.]*

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), 1981. *[page 18.]*

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020. *[pages 43 and 33.]*

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. Cryptology ePrint Archive, Report 2002/140, 2002. https://eprint.iacr.org/2002/140. *[pages 10, 35, and 26.]*

[CP93] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993. *[page 19.]*

[DEF+23] Poulami Das, Andreas Erwig, Sebastian Faust, Julian Loss, and Siavash Riahi. BIP32-compatible threshold wallets. Cryptology ePrint Archive, Report 2023/312, 2023. https://eprint.iacr.org/2023/312. *[pages 21 and 12.]*

[DFL19] Poulami Das, Sebastian Faust, and Julian Loss. A formal treatment of deterministic wallets. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 651–668. ACM Press, November 2019. *[pages 21 and 12.]*

[DJN+20] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergård. Fast threshold ECDSA with honest majority. Cryptology ePrint Archive, Report 2020/501, 2020. https://eprint.iacr.org/2020/501. *[pages 22 and 14.]*

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR TCHES*, 2018(1):238–268, 2018. https://tches.iacr.org/index.php/TCHES/article/view/839. *[page 18.]*

[DKLs18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, pages 980–997. IEEE Computer Society Press, May 2018. *[pages 22 and 14.]*

[Edg23] Ben Edgington. *Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond.* 2023. https://eth2book.info/latest/. *[pages 22 and 14.]*

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 10–18. Springer, Heidelberg, August 1984. *[pages 25 and 17.]*

[ER22] Andreas Erwig and Siavash Riahi. Deterministic wallets for adaptor signatures. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part II*, volume 13555 of *LNCS*, pages 487–506. Springer, Heidelberg, September 2022. *[pages 21 and 12.]*

[Eth23a] Ethereum. Account Abstraction. ethereum.org/en/roadmap/account-abstraction, 2023. *[page 18.]*

[Eth23b] Ethereum. Optimistic rollups. https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/, 2023. *[page 13.]*

[Eya21] Ittay Eyal. On cryptocurrency wallet design. Cryptology ePrint Archive, Report 2021/1522, 2021. https://eprint.iacr.org/2021/1522. *[pages 21 and 12.]*

[FK97] Uriel Feige and Joe Kilian. Making games short (extended abstract). In *29th ACM STOC*, pages 506–516. ACM Press, May 1997. *[page 13.]*

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987. *[pages 9, 27, and 18.]*

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1179–1194. ACM Press, October 2018. *[pages 22 and 14.]*

[GGJ$^+$24] Sanjam Garg, Noemi Glaeser, Abhishek Jain, Michael Lodder, and Hart Montgomery. Hot-cold threshold wallets with proofs of remembrance, 2024. *[page 20.]*

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. Cryptology ePrint Archive, Report 2016/013, 2016. https://eprint.iacr.org/2016/013. *[pages 22 and 14.]*

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, Heidelberg, August 2010. *[page 12.]*

[GKSŚ20] Adam Gągol, Jędrzej Kula, Damian Straszak, and Michał Świętek. Threshold ECDSA for decentralized asset custody. Cryptology ePrint Archive, Report 2020/498, 2020. https://eprint.iacr.org/2020/498. *[pages 22 and 14.]*

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016. *[page 15.]*

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953. *[page 17.]*

[Hu23] Mingxing Hu. Post-quantum secure deterministic wallet: Stateless, hot/cold setting, and more secure. Cryptology ePrint Archive, Report 2023/062, 2023. https://eprint.iacr.org/2023/062. *[pages 21 and 12.]*

[IZ89] Russell Impagliazzo and David Zuckerman. How to recycle random bits. In *30th FOCS*, pages 248–253. IEEE Computer Society Press, October / November 1989. *[pages 10 and 9.]*

[KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, Heidelberg, October 2020. *[pages 22 and 14.]*

[KMOS21] Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. Refresh when you wake up: Proactive threshold wallets with offline devices. In *2021 IEEE Symposium on Security and Privacy*, pages 608–625. IEEE Computer Society Press, May 2021. *[pages 21, 23, 12, and 14.]*

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010. *[pages 8 and 20.]*

[Lab23] Offchain Labs. Inside Arbitrum Nitro. `https://docs.arbitrum.io/inside-arbitrum-nitro/`, 2023. *[page 13.]*

[LN18] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1837–1854. ACM Press, October 2018. *[pages 22 and 14.]*

[LS91] Dror Lapidot and Adi Shamir. Publicly verifiable non-interactive zero-knowledge proofs. In Alfred J. Menezes and Scott A. Vanstone, editors, *CRYPTO'90*, volume 537 of *LNCS*, pages 353–365. Springer, Heidelberg, August 1991. *[page 16.]*

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988. *[page 20.]*

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008. *[pages 5 and 15.]*

[Oka93] Tatsuaki Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 31–53. Springer, Heidelberg, August 1993. *[page 9.]*

[Opt23] Optimism. Rollup Protocol. `https://community.optimism.io/docs/protocol/2-rollup-protocol`, 2023. *[page 13.]*

[PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017. *[page 14.]*

[PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. `https://lightning.network/lightning-network-paper.pdf`, 2016. *[page 14.]*

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, August 1992. *[pages 9 and 8.]*

[Pet21] Michaella Pettit. Efficient threshold-optimal ECDSA. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *CANS 21*, volume 13099 of *LNCS*, pages 116–135. Springer, Heidelberg, December 2021. *[pages 22 and 14.]*

[PFH$^+$22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. *[page 18.]*

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019. *[page 17.]*

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990. *[pages 15, 27, and 19.]*

[SGB24] István András Seres, Noemi Glaeser, and Joseph Bonneau. Short paper: Naysayer proofs. In Jeremy Clark and Elaine Shi, editors, *Financial Cryptography and Data Security*, 2024. *[page 12.]*

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979. *[page 6.]*

[SJSW19] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. ETHDKG: Distributed key generation with Ethereum smart contracts. Cryptology ePrint Archive, Report 2019/985, 2019. https://eprint.iacr.org/2019/985. *[page 14.]*

[SNBB19] István András Seres, Dániel A. Nagy, Chris Buckland, and Péter Burcsi. MixEth: efficient, trustless coin mixing service for Ethereum. Cryptology ePrint Archive, Report 2019/341, 2019. https://eprint.iacr.org/2019/341. *[pages 14 and 18.]*

[SV05] Nigel Smart and Frederik Vercauteren. On computable isomorphisms in efficient asymmetric pairing based systems. Cryptology ePrint Archive, Report 2005/116, 2005. https://eprint.iacr.org/2005/116. *[page 7.]*

[Tha23] Justin Thaler. Proofs, arguments, and zero-knowledge, July 2023. *[page 6.]*

[TR19] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *arXiv preprint arXiv:1908.04756*, 2019. *[page 13.]*

[Woo24] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, june 2024. *[page 12.]*

[Wui12] Pieter Wuille. BIP32: Hierarchical deterministic wallets. https://en.bitcoin.it/wiki/BIP_0032, February 2012. *[pages 21 and 12.]*

[ZBK$^+$22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3121–3134. ACM Press, November 2022. *[pages 28 and 19.]*

Share Refresh Proofs (Π_{Ref})

Parameters: Degree-d KZG common reference string $\mathsf{crs} = \{g_1, g_1^{\tau}, \ldots, g_1^{\tau^d}, g_2, g_2^{\tau}\}$.

$\underline{\mathsf{Prove}((\mathsf{crs}, \mathsf{ucom}_T, t-1, \{\delta_i\}_{i\in[n]}); z_T(X)) \to (\{\zeta_{T,i}\}_{i\in[n]}, \pi_z)}$: Given crs, a KZG commitment ucom_T to update polynomial $z_T(X)$, the latter's degree $t-1$, and each party's share refresh information δ_i, use $z_T(X)$ to compute the following:

1. For each $i \in [n]$, prove that $\delta_i = z_T(i)$ by computing $(\delta_i, \zeta_{T,i}) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, z_T(X), i)$.
2. Prove that $z_T(0) = 0$ and $z_T(X)$ has degree $t-1 \le d$ by computing $(0, \zeta_{T,0}) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, z_T(X), 0)$ and $\mathsf{dcom}_T := g_1^{\tau^{d-t+1}\cdot z_t(\tau)}$ using crs. Let $\pi_{\mathsf{ucom}} := (\zeta_{T,0}, \mathsf{dcom}_T)$.
3. Output $(\{\zeta_{T,i}\}_{i\in[n]}, \pi_{\mathsf{ucom}})$.

$\underline{\mathsf{HVrfy}((\mathsf{crs}, \mathsf{ucom}_T, i, \delta_i), \zeta_{T,i}) \to \{0,1\}}$: Given crs, a KZG commitment ucom_T, party index i, and share refresh information δ_i, output $\mathsf{KZG.Vrfy}(\mathsf{crs}, \mathsf{ucom}_T, i, \delta_i, \zeta_{T,i})$.

$\underline{\mathsf{UCVrfy}((\mathsf{crs}, \mathsf{ucom}_T, t-1), \pi_{\mathsf{ucom}}) \to \{0,1\}}$: Given crs, a KZG commitment ucom_T, and its supposed degree $t-1$, verify the proof $\pi_{\mathsf{ucom}} = (\zeta_{T,0}, \mathsf{dcom}_T)$ by outputting 1 iff the following hold:

$$\mathsf{KZG.Vrfy}(\mathsf{crs}, \mathsf{ucom}_T, 0, 0, \zeta_{T,0}) = 1$$
$$e(\mathsf{dcom}_T, g_2) = e(\mathsf{ucom}_T, g_2^{d-t+1})$$

Figure 11: The proof system Π_{Ref} used to prove correctness of the every hot party's share refresh information δ_i and the commitment update ucom_t. Each hot party verifies its own update information using HVrfy, and the correctness of ucom is verified separately via UCVrfy.

Public parameters: Groups $\mathbb{G}_1, \mathbb{G}_2$ of order p with generators g_1, g_2, respectively.

- On input $(\texttt{sid}, \mathsf{Setup}, C, (\mathsf{vk}, t, \mathsf{com}))$, delete any existing entries $(C, *, *, *, *) \in D$ and $(C, *, *) \in U$. Add $(C, \mathsf{vk}, t, \mathsf{com}, \texttt{time} := 0)$ to D and output $(\texttt{sid}, \mathsf{SetupResult}, C, (1))$.
- On input $(\texttt{sid}, \mathsf{ComUpdate}, C, (\mathsf{ucom}, \pi_{\mathsf{ucom}}))$:
  1. Retrieve $(C, \mathsf{vk}, t, \mathsf{com}, \texttt{time}) \in D$ for the maximum value of $\texttt{time}$.
  2. If $\Pi_{\mathsf{Ref}}.\mathsf{UCVrfy}((\mathsf{crs}, \mathsf{ucom}, t), \pi_{\mathsf{ucom}}) = 1$, set $b := 1$ and add $(C, \mathsf{vk}, t, \mathsf{com} \cdot \mathsf{ucom}, \texttt{time++})$ to D and $(C, \mathsf{ucom}, \texttt{time++})$ to U. Otherwise set $b := 0$.
  3. Output $(\texttt{sid}, \mathsf{ComUpdateResult}, C, (b))$.
- On input $(\texttt{sid}, \mathsf{ClientInfoRecover}, P, (C))$, retrieve $(C, \mathsf{vk}, *, \mathsf{com}, \texttt{time}) \in D$ for the maximum value of $\texttt{time}$ and output $(\texttt{sid}, \mathsf{ClientInfo}, P, (C, \mathsf{vk}, \mathsf{com}))$.
- On input $(\texttt{sid}, \mathsf{UComRecover}, P, (C))$, retrieve $(C, \mathsf{ucom}, \texttt{time}) \in U$ for the maximum value of $\texttt{time}$ and output $(\texttt{sid}, \mathsf{UCom}, P, (C, \mathsf{ucom}))$.

Figure 12: **The Programmable Bulletin-Board Functionality** $\mathcal{F}_{\mathsf{BB}}$

Encrypted secret sharing protocol

Public parameters: Groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p with generators g_1, g_2, respectively; a degree-d KZG common reference string crs.

Setup: On input $(\mathtt{sid}, \mathsf{SSSetup}, C, (t, \mathcal{P}, \{\mathsf{ek}_i\}_{i \in [n]}))$, where $t, n \in \mathbb{N}$ s.t. $n = |\mathcal{P}|$ and $t \leq n \leq d$, $\mathcal{P} = \{P_1 \dots P_n\}$ a set of parties, and $\{\mathsf{ek}_i\}_{i \in [n]}$ a set of public (encryption) keys $(\forall i \in [n], \mathsf{ek}_i \in \mathbb{G}_2)$, C proceeds as follows:

1. Sample $x \xleftarrow{\$} \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
2. Generate t-out-of-n Shamir Shares of x as $x_1, \dots x_n \in \mathbb{Z}_p$. Let $y_i := g_2^{x_i}$ $\forall i \in [n]$.
3. Interpolate the degree-n polynomial $\tilde{f}$ such that $\tilde{f}(i) = \mathcal{H}(\mathsf{ek}_i^x) + x_i$ $\forall i \in [n]$. Compute $\mathsf{com} \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, \tilde{f})$ and send $(\mathtt{sid}, \mathsf{Setup}, C, (y, t, n, \mathsf{com}))$ to $\mathcal{F}_{\mathsf{BB}}$.
4. Delete any existing entries $(C, *, *) \in D$ and add $(C, \mathcal{P}, t)$ to D.
5. For each $i \in [n]$, compute $(\tilde{x}_i, \pi_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, \tilde{f}, i)$ and output $(\mathtt{sid}, \mathsf{SecretShare}, P_i, (C, i, \tilde{x}_i, \pi_i))$.
6. Finally, output $(\mathtt{sid}, \mathsf{SSSetupDone}, C, (y, \{y_i\}_{i \in [n]}))$.

Generating share refreshes: On input $(\mathtt{sid}, \mathsf{ZeroSetup}, C, (t, \mathcal{P}))$, where $t, n \in \mathbb{N}$ s.t. $n = |\mathcal{P}|$ and $t \leq n \leq d$, and $\mathcal{P} = \{P_1 \dots P_n\}$ a set of parties, C will proceed as follows:

1. Generate t-out-of-n Shamir Shares of 0 as $x_1, \dots x_n \in \mathbb{Z}_p$; let f be the polynomial used.
2. Compute $\mathsf{com}_0 \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, f)$ and $(\{\zeta_i\}_{i \in [n]}, \pi_z) \leftarrow \Pi_{\mathsf{Ref}}.\mathsf{Prove}((\mathsf{crs}, \mathsf{com}_0, t - 1, \{x_i\}_{i \in [n]}); f(X))$. Send $(\mathtt{sid}, \mathsf{ComUpdate}, C, (\mathsf{com}_0, \pi_z))$ to $\mathcal{F}_{\mathsf{BB}}$, which returns $(\mathtt{sid}, \mathsf{ComUpdateResult}, C, (b))$.
3. Send (x_i, ζ_i) to P_i for all $i \in [n]$, then output $(\mathtt{sid}, \mathsf{ZeroSetupDone}, C, (b))$.
4. Each party P_i for $i \in [n]$ will send $(\mathtt{sid}, \mathsf{UComRecover}, P_i, (C))$ to $\mathcal{F}_{\mathsf{BB}}$ and receive $(\mathtt{sid}, \mathsf{UCom}, P_i, (C, \mathsf{ucom}))$ in return. Check that $\Pi_{\mathsf{Ref}}.\mathsf{HVrfy}((\mathsf{crs}, \mathsf{ucom}, i, x_i), \zeta_i) = 1$. If not, set $x_i, \zeta_i = \perp$. Output $(\mathtt{sid}, \mathsf{ZeroShare}, P_i, (C, x_i, \zeta_i))$.

Providing auxiliary information: On input $(\mathtt{sid}, \mathsf{AuxRecover}, P, (C))$ for some client C, P sends $(\mathtt{sid}, \mathsf{ClientInfoRecover}, C)$ to $\mathcal{F}_{\mathsf{BB}}$ to get $(\mathsf{vk}, \mathsf{com})$. It outputs $(\mathtt{sid}, \mathsf{AuxInfo}, P, (C, \mathsf{vk}, \mathsf{com}))$.

Figure 13: Protocol realizing $\mathcal{F}_{\mathsf{SS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model