Practical Cryptography for Blockchains: Secure Protocols with Minimal Trust

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. This growing popularity has put pressure on the ecosystem to prioritize scalability at the expense of trustlessness and decentralization.

This work explores the role cryptography has to play in the blockchain ecosystem to improve performance while maintaining minimal trust and strong security guarantees. I discuss a new paradigm for scalability, called *naysayer proofs*, which sits between optimistic and zero-knowledge approaches. Next, I show how to construct efficient, non-interactive, and fair on-chain protocols for a large class of elections and auctions without a trusted third party. Finally, I introduce a new type of threshold wallet which adds offline components at each custodian to improve security while maintaining distributed trust. 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 in applying them. These examples show that advanced cryptography has the potential to meaningfully nudge the blockchain ecosystem towards increased security and reduced trust.

Acknowledgments

Alex Block and Doruk Gür for feedback on naysayer proofs for FRI (Section 3.4.2) and Dilithium (Section 3.4.3), respectively.

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. Available at https://eprint.iacr.org/2023/1473.

[GSZB24] Noemi Glaeser, István András Seres, Michael Zhu, and Joseph Bonneau. Cicada: A framework for private non-interactive on-chain auctions and voting. In *Workshop on Cryptographic Tools for Blockchains*, 2024.

[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.

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.

Unfortunately, as the number of applications and their users has increased, developers have been forced to sacrifice trustlessness and sometimes even security or privacy in favor of performance and scalability. This dissertation uses cryptographic tools to enable blockchain applications which are both *practical* and *secure* while staying true to the original blockchain ethos and minimizing trust. All the works discussed were created in collaboration with industry practitioners and have seen interest or deployment in the blockchain ecosystem.

In Section 2, I introduce the necessary background and building blocks used throughout this dissertation. Section 3 describes *naysayer proofs*, a new paradigm for verifying zero-knowledge proofs in the so-called optimistic setting. One notable application of naysayer proofs is for scalability, where it sits between the existing solutions—optimistic rollups and zero-knowledge rollups—and can provide better performance and accessibility for certain parties in rollup ecosystems. Since their publication, naysayer proofs have seen interest in production deployment from at least two startups (that I am aware of) in the blockchain space.

In Section 4, we move to on-chain applications and describe Cicada, a smart contract protocol for realizing the first *fair* and *non-interactive* on-chain elections and auctions. This resolves important security and usability hurdles in existing systems, which are widely used for on-chain governance votes or sales of digital goods such as non-fungible tokens (NFTs). Cicada is accompanied by a Solidity smart contract implementation, making it easily deployable on Ethereum and a large number of "layer 2" (L2) chains. Furthermore, we have been in contact with Optimism, one of the largest L2s in the ecosystem, to discuss the use of Cicada for their retroPGF funding vote.[1]

Finally, Section 5 moves off-chain and looks at securing the edges of the system: cryptocurrency wallets. We propose a new architecture which aims to boost the security of threshold wallets while maintaining their usability benefits, and present constructions for two popular signature schemes. This design was created with collaborators from Lit Protocol[2] and the Linux Foundation. An Apache 2.0-licensed is also publicly available in Hyperledger Labs[3], a popular

[1] https://community.optimism.io/docs/governance/
[2] https://www.litprotocol.com/
[3] https://github.com/hyperledger-labs

open source organization.

We conclude in Section 6 by discussing potential future directions for these three works and a broader outlook on the role of cryptography in blockchains.

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 $z_i = x_i + y_i \ \forall i \in [\ell]$. todo: empty set $\varnothing$ vs. $\perp$ Sampling an element x uniformly at random from a set $\mathcal{X}$ is written as $x \leftarrow\!\!\$\ \mathcal{X}$. 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 $y \leftarrow\!\!\$\ \mathsf{Alg}(x)$ if the algorithm is randomized (or sometimes $\mathsf{Alg}(x) \$\!\!\rightarrow y$). When we wish to be explicit about the randomness r used, we write $y \leftarrow \mathsf{Alg}(x; r)$. We use $\mathcal{D}_1 \approx_\lambda \mathcal{D}_2$ to denote that two distributions $\mathcal{D}_1, \mathcal{D}_2$ have statistical distance bounded by $\mathsf{negl}(\lambda)$.

For a non-interactive 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

todo: NP-languages and relations Noemi: A language is some subset of strings. Every NP language has a corresponding polynomially-decidable relation (i.e., decidable by a circuit $C(x, w)$ such that $|C| \in \mathsf{poly}(|x|)$) such that if $x \in \mathcal{L}, \exists w$ s.t. $(x, w) \in \mathcal{R}_\mathcal{L}$ where w is polynomial in $|x|$.

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

- $\mathsf{Setup}(1^\lambda) \to \mathsf{crs}$*: Given a security parameter, output a common reference string* crs*. This algorithm might use private randomness (a trusted setup).*
- $\mathsf{Prove}(\mathsf{crs}, x, w) \to \pi$*: Given the* crs*, an instance* x*, and witness* w *such that* $(x, w) \in \mathcal{R}_\mathcal{L}$*, 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 require (perfect) completeness*, meaning that for all* $(x, w) \in \mathcal{R}_L ang$*,*

$$\Pr\left[\mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 1 \;\middle|\; \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \wedge \\ \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, x, w) \end{array}\right] = 1.$$

Definition 2 (soundness)**.** *Soundness requires that for all $x \notin \mathcal{L}$, $\lambda \in \mathbb{N}$, and all PPT adversaries $\mathcal{A}$,*

$$\Pr\left[\mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 1 \;\middle|\; \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \wedge \\ \pi \leftarrow \mathcal{A}(\mathsf{crs}, x) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We refer the reader to [Tha23, Gol01] for a formal description of other properties of proof systems (e.g., correctness, zero-knowledge).

There are numerous definitions of succinctness adopted for proof systems in the literature. In this work, we require succinct proof systems only to have proofs which are sublinear in $|w|$ and "work-saving" verification (faster than $C(x, w)$)[4]:

Definition 3 (succinct)**.** *We say a proof system Π is* succinct *if $|\pi| \in o(|w|)$ and $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi)$ runs in time $o(\mathsf{poly}(\lambda)|x|)$.*

2.2 Time-Lock Puzzles

A time-lock puzzle (TLP) [RSW96] consists of three efficient algorithms $\mathsf{TLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve})$ allowing a party to "encrypt" a message to the future. To recover the solution, one needs to perform a computation that is believed to be inherently sequential, with a parameterizable number of steps.

Definition 4 (Time-lock puzzle [RSW96])**.** *A time-lock puzzle scheme $\mathsf{TLP} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve})$ for solution space $\mathcal{X}$ consists of the following three efficient algorithms:*

- $\mathsf{TLP.Setup}(1^\lambda, T) \xrightarrow{\$} \mathsf{pp}$*: The (potentially trusted) setup algorithm takes as input a security parameter 1^λ and a difficulty (time) parameter T, and outputs public parameters* pp*.*
- $\mathsf{TLP.Gen}(\mathsf{pp}, s) \xrightarrow{\$} Z$*: Given a solution $s \in \mathcal{X}$, the puzzle generation algorithm efficiently computes a time-lock puzzle Z.*
- $\mathsf{TLP.Solve}(\mathsf{pp}, Z) \to s$*: Given a TLP Z, the puzzle-solving algorithm requires at least T sequential steps to output the solution s.*

Informally, we say that a TLP scheme is *correct* if $\mathsf{TLP.Gen}$ is efficiently computable, and $\mathsf{TLP.Solve}$ always recovers the original solution s to a validly constructed puzzle. A TLP scheme is *secure* if Z hides the solution s and no adversary can compute $\mathsf{TLP.Solve}$ in fewer than T steps with non-negligible probability. For the formal definitions, see [MT19].

[4] https://a16zcrypto.com/posts/article/17-misconceptions-about-snarks/#section--3

Homomorphic TLPs. Malavolta and Thyagarajan [MT19] introduce *homomorphic* TLPs (HTLPs). An HTLP is defined with respect to a circuit class $\mathcal{C}$ and has an additional algorithm Eval defined as:

- $\underline{\mathsf{HTLP.Eval}(\mathsf{pp}, C, Z_1, \ldots, Z_m) \to Z_*}$: Given the public parameters, a circuit $C \in \mathcal{C}$ where $C : \mathcal{X}^m \to \mathcal{X}$, and input puzzles $Z_1, \ldots, Z_m$, the homomorphic evaluation algorithm outputs a puzzle Z_*.

Correctness requires that $\mathsf{HTLP.Solve}(Z_*)$ should contain the expected solution, namely $C(s_1, \ldots, s_m)$, where $s_i \leftarrow \mathsf{HTLP.Solve}(Z_i)$. Again, we refer the reader to [MT19] for the formal definition. Moving forward, we will use $\boxplus$ for homomorphic addition and $\cdot$ for scalar multiplication of HTLPs. For the homomorphic application of a linear function f, we write $f(Z_1, \ldots, Z_m)$.

2.3 HTLP Constructions

Malavolta and Thyagarajan [MT19] give two HTLP constructions with linear and multiplicative homomorphisms, respectively. They require N to be a *strong* semiprime, i.e., $N = p \cdot q$ such that $p = 2p' + 1$ and $q = 2q' + 1$ where p', q' are also prime. The linearly-homomorphic HTLP is based on Paillier encryption [**?**], while the multiplicative homomorphism is achieved by working over the subgroup $\mathbb{J}_N \subseteq \mathbb{Z}_N^*$ of elements with Jacobi symbol $+1$. We recall their constructions below.

Construction 1 (Linear HTLP [MT19].)**.**

- $\underline{\mathsf{HTLP.Setup}(1^\lambda, T) \xrightarrow{\$} \mathsf{pp}}$*: Sample a strong semiprime N and a generator $g \xleftarrow{\$} \mathbb{Z}_N^*$, then compute $h = g^{2^T} \mod N \in \mathbb{Z}_N^*$. (This can be computed efficiently using the factorization of N). Output* $\mathsf{pp} := (N, g, h)$.
- $\underline{\mathsf{HTLP.Gen}(\mathsf{pp}, s; r) \to Z}$*: Given a value $s \in \mathbb{Z}_N$, use randomness $r \in \mathbb{Z}_{N^2}$ to compute and output*

$$Z := (g^r \mod N,\ h^{r \cdot N} \cdot (1+N)^s \mod N^2) \in \mathbb{J}_N \times \mathbb{Z}_{N^2}^*$$

- $\underline{\mathsf{HTLP.Open}(\mathsf{pp}, Z, r) \to s}$*: Parse $Z := (u, v)$ and compute $w := u^{2^T} \mod N = h^r$ via repeated squaring. Output $s := \frac{(v/w^N \mod N^2) - 1}{N}$.*
- $\underline{\mathsf{HTLP.Eval}(\mathsf{pp}, f, Z_1, Z_2) \to Z}$*: To evaluate a linear function $f(x_1, x_2) = b + a_1 x_1 + a_2 x_2$ homomorphically on puzzles $Z_1 := (u_1, v_1)$ and $Z_2 := (u_2, v_2)$, return*

$$Z = (u_1^{a_1} \cdot u_2^{a_2} \mod N, v_1^{a_1} \cdot v_2^{a_2} \cdot (1+N)^b \mod N^2).$$

Correctness holds because for all $s \in \mathbb{Z}_N$ and $Z = (u, v) \leftarrow \mathsf{HTLP.Gen}(\mathsf{pp}, s)$,

$$\mathsf{HTLP.Open}(\mathsf{pp}, Z) = \frac{(v/(h^R)^N \mod N^2) - 1}{N} = \frac{((1+N)^s) - 1}{N} = s \qquad (1)$$

since $(1+N)^x = 1 + Nx \mod N^2$. Correctness of the homomorphism follows since for all linear functions $f(x_1, x_2) = b + a_1 x_1 + a_2 x_2$ and all $Z_i = (u_i, v_i) \in \mathsf{Im}(\mathsf{HTLP.Gen}(\mathsf{pp}, s_i; r_i))$ for $i \in \{1, 2\}$,[5]

$$\begin{aligned}
&\mathsf{HTLP.Eval}(\mathsf{pp}, f, Z_1, Z_2) = (u_1^{a_1} \cdot u_2^{a_2}, (1+N)^b \cdot v_1^{a_1} \cdot v_2^{a_2}) \\
&= (g^{r_1 a_1} \cdot g^{r_2 a_2}, \quad (1+N)^b \cdot h^{r_1 N a_1} \cdot (1+N)^{s_1 a_1} \cdot h^{r_2 N a_2} \cdot (1+N)^{s_2 a_2}) \\
&= (g^{r_1 a_1 + r_2 a_2}, \quad h^{(r_1 a_1 + r_2 a_2) \cdot N} \cdot (1+N)^{b + s_1 a_1 + s_2 a_2}) \\
&= \mathsf{HTLP.Gen}(\mathsf{pp}, f(s_1, s_2); r_1 a_1 + r_2 a_2)
\end{aligned}$$

which opens to $f(s_1, s_2)$ by eq. (1).

Construction 2 (Multiplicative HTLP [MT19].)**.**

- $\mathsf{HTLP.Setup}(1^\lambda, T) \xrightarrow{\$} \mathsf{pp}$*: Same as Construction 1.*
- $\mathsf{HTLP.Gen}(\mathsf{pp}, s; r) \to Z$*: Given a value* $s \in \mathbb{J}_N$*, use randomness* $r \in \mathbb{Z}_{N^2}$ *to compute and output*

$$Z := (g^r \mod N,\ h^r \cdot s \mod N) \in \mathbb{Z}_N^* \times \mathbb{Z}_N^*$$

- $\mathsf{HTLP.Open}(\mathsf{pp}, Z, r) \to s$*: Parse* $Z := (u, v)$ *and compute* $w := u^{2^T} \mod N = h^r$ *via repeated squaring. Output* $s := v/w$*.*
- $\mathsf{HTLP.Eval}(\mathsf{pp}, f, Z_1, Z_2) \to Z$*: To evaluate a multiplicative function* $f(x_1, x_2) = a x_1 x_2$ *homomorphically on puzzles* $Z_1 := (u_1, v_1)$ *and* $Z_2 := (u_2, v_2)$*, return*

$$Z = (u_1 \cdot u_2 \mod N, a \cdot v_1 \cdot v_2 \mod N)$$

Construction 2 operates over the solution space $\mathbb{J}_N$ (instead of $\mathbb{Z}_N$). It is easy to see that $\mathsf{HTLP.Open}(\mathsf{pp}, \mathsf{HTLP.Gen}(\mathsf{pp}, s)) = s$ for all $s \in \mathbb{Z}_N^*$. Furthermore, for all $f(x_1, x_2) = a x_1 x_2$ and all $Z_i = (u_i, v_i) \in \mathsf{Im}(\mathsf{HTLP.Gen}(\mathsf{pp}, s_i; r_i))$ for $i \in \{1, 2\}$,

$$\begin{aligned}
\mathsf{HTLP.Eval}(\mathsf{pp}, f, Z_1, Z_2) &= (u_1 \cdot u_2 \mod N, a \cdot v_1 \cdot v_2 \mod N) \\
&= (g^{r_1} g^{r_2} \mod N, \quad h^{r_1} h^{r_2} \cdot a s_1 s_2 \mod N) \\
&= (g^{r_1 + r_2} \mod N, \quad h^{r_1 + r_2} \cdot a s_1 s_2 \mod N) \\
&= \mathsf{HTLP.Gen}(\mathsf{pp}, f(s_1, s_2); r_1 + r_2)
\end{aligned}$$

2.4 Vector Packing

Definition 5 (Vector packing)**.** *A setup algorithm* PSetup *and pair of efficiently computable bijective functions* $(\mathsf{Pack}, \mathsf{Unpack})$ *is called a* packing scheme *and has the following syntax:*

[5] For space and clarity we drop the moduli and assume that we are working in the appropriate ring in each coordinate (namely $\mathbb{Z}_N$ and $\mathbb{Z}_{N^2}$, respectively).

- $\underline{\mathsf{PSetup}(\ell, w) \to \mathsf{pp}}$*: Given a vector dimension* ℓ *and maximum entry* w*, output public parameters* pp*.*
- $\underline{\mathsf{Pack}(\mathsf{pp}, \mathbf{a}) \to s}$*: Encode* $\mathbf{a} \in (\mathbb{Z}^+)^\ell$ *as a positive integer* $s \in \mathbb{Z}^+$*.*
- $\underline{\mathsf{Unpack}(\mathsf{pp}, s) \to \mathbf{a}}$*: Given* $s \in \mathbb{Z}^+$*, recover a vector* $\mathbf{a} \in (\mathbb{Z}^+)^\ell$*.*

For correctness *we require* $\mathsf{Unpack}(\mathsf{Pack}(\mathbf{a})) = \mathbf{a}$ *for all* $\mathbf{a} \in (\mathbb{Z}^+)^\ell$*.*

The classic approach to packing [Gro05, HS00] uses a *positional numeral system (PNS)* to encode a vector of entries bounded by w as a single integer in base $M := w$ (see Construction 3 below). Instead, we will set $M := nw + 1$ to accommodate the homomorphic addition of all n users' vectors: each voter submits a length-m vector with entries $\leq w$. Summing over n voters, the result is a length-m vector with a maximum entry value nw; to prevent overflow, we set $M = nw + 1$.

Construction 3 (Packing from Positional Numeral System)**.**

- $\underline{\mathsf{PSetup}(\ell, w) \to M}$*: Return* $M := w + 1$*.*
- $\underline{\mathsf{Pack}(M, \mathbf{a}) \to s}$*: Output* $s := \sum_{j=1}^{|\mathbf{a}|} a_j M^{j-1}$*.*
- $\underline{\mathsf{Unpack}(M, s) \to \mathbf{a}}$*: Let* $\ell := \lceil \log_M s \rceil$*. For* $j \in [\ell]$*, compute the* j*th entry of* $\mathbf{a}$ *as* $a_j := s \mod M^{j-1}$*.*

Besides PNS packing, we also introduce an alternative approach in Construction 4 which is based on the *residue numeral system (RNS)*. The idea of the RNS packing is to interpret the entries of $\mathbf{a}$ as prime residues of a single unique integer s, which can be found efficiently using the Chinese Remainder Theorem (CRT). In other words, for all $j \in [\ell]$, s captures a_j as $s \bmod p_j$.

Construction 4 (Packing from Residue Numeral System)**.**

- $\underline{\mathsf{PSetup}(\ell, w) \to \mathbf{p}}$*: Let* $M := w+1$ *and sample* ℓ *distinct primes* $p_1, \ldots, p_\ell$ *s.t.* $p_j \geq M \ \forall j \in [\ell]$*. Return* $\mathbf{p} := (p_1, \ldots, p_\ell)$*.*
- $\underline{\mathsf{Pack}(\mathbf{p}, \mathbf{a}) \to s}$*: Given* $\mathbf{a} \in (\mathbb{Z}^+)^\ell$*, use the CRT to find the unique* $s \in \mathbb{Z}^+$ *s.t.* $s \equiv a_j \pmod{p_j} \ \forall j \in [\ell]$*.*
- $\underline{\mathsf{Unpack}(\mathbf{p}, s) \to \mathbf{a}}$*: return* $(a_1, \ldots, a_\ell)$ *where* $a_j \equiv s \mod p_j \ \forall j \in [\ell]$*.*

A major advantage of this approach is that, in contrast to the PNS approach, which is only homomorphic for SIMD (single instruction, multiple data) addition, the RNS encoding is fully SIMD homomorphic: the sum of vector encodings $\sum_{i \in [n]} s_i$ encodes the vector $\mathbf{a}_+ = \sum_{i \in [n]} \mathbf{a}_i$, and the product $\prod_{i \in [n]} s_i$ encodes the vector $\mathbf{a}_\times = \prod_{i \in [n]} \mathbf{a}_i$.

2.5 Bilinear Pairings

Definition 6 (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:*

$$e(g_1^a, g_2) = e(g_1, g_2^a) = e(g_1, g_2)^a$$
$$e(g_1 h_1, g_2) = e(g_1, g_2) \cdot e(h_1, g_2)$$
$$e(g_1, g_2 h_2) = e(g_1, g_2) \cdot e(g_1, h_2)$$

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.*

2.6 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 5 (Shamir secret sharing [Sha79])**.** *Let* p *be a prime.*

- $\{s_1, \ldots, s_n\} \leftarrow \mathsf{Share}(s, t, n)$*: 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* $s_i := (i, f(i))$*.*
- $\{s', \perp\} \leftarrow \mathsf{Reconstruct}(S, t, n)$*: Given some set of shares* S*, check if* $|S| < t$*. If so, output* $\perp$*. 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:

- $\{s_k, \perp\} \leftarrow \mathsf{Interpolate}(S, k, t, n)$: 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 $s_k := (k, f(k))$.

2.7 Digital Signatures

A *digital signature* todo: [?] is a cryptographic primitive which todo: ...

2.7.1 BLS Signatures

Construction 6 (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:*

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda)$*: Given the security parameter 1^λ, sample $x \leftarrow\!\!\$\ \mathbb{Z}_p$ and output the keypair consisting of signing key $\mathsf{sk} := x$ and verification key $\mathsf{vk} := g_2^x$.*
- $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$*: Given a signing key $\mathsf{sk} \in \mathbb{Z}_p$ and a message $m \in \{0,1\}^*$, compute a signature $\sigma := H(m)^{sk}$.*
- $\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{vk}, m, \sigma)$*: 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:

- $\sigma \leftarrow \mathsf{Reconstruct}(S := \{(i, \sigma_i)\})$: Let $S' \subseteq S$ be the set of valid partial signatures in S. If $|S'| < t$, output $\perp$. Otherwise, without loss of generality, assume the first t valid signatures come from users $1, \ldots, t$ 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.

2.7.2 Schnorr Signatures

Construction 7 (Schnorr signature [Sch90])**.** *Let* $\mathsf{pp} = (\mathbb{G}, g, q)$ *be description of a cyclic group* $\mathbb{G}$ *of prime order* q *with generator* g *in which the discrete logarithm problem is hard.*

- $\mathsf{KGen}(\mathsf{pp}) \to (\mathsf{sk}, \mathsf{vk})$*: Sample a secret key* $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_q$ *and set the corresponding verification key* $\mathsf{vk} := g^{\mathsf{sk}} \in \mathbb{G}$*. Output* $(\mathsf{sk}, \mathsf{vk})$*.*
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$*: Given a secret key* $\mathsf{sk} \in \mathbb{Z}_q$ *and message* m*, sample* $k \leftarrow\!\!\$\ \mathbb{Z}_q$ *and let* $R := g^k$*. Compute* $c := H(m, g^{\mathsf{sk}}, R)$ *and* $s := k + c \cdot \mathsf{sk}$*. Output* $\sigma := (R, s)$*.*
- $\mathsf{Vrfy}(\mathsf{vk}, m, \sigma) \to \{0,1\}$*: Given a verification key* vk*, message* m*, and signature* σ*, compute* $c := H(m, \mathsf{vk}, R)$ *and check* $R \cdot \mathsf{vk}^c \stackrel{?}{=} s$*. If so, output 1, else 0.*

Noemi: Schnorr (threshold) signatures; [Lin22] and [KG20] use different definitions of Schnorr, are they equivalent?

Threshold variants Many protocols exist for threshold Schnorr signatures todo: [?], each offering a different combination of properties. todo: ...

The BLS threshold signature scheme [BLS01] is fully non-interactive, meaning that signing occurs in one round. FROST is a "somewhat" non-interactive threshold signature scheme [BTZ22] in the sense that the *signing* procedure consists of a single round, but there is another round of message-independent pre-processing.

2.8 KZG polynomial commitments

KZG commitments can be instantiated using either a symmetric or asymmetric pairing; we give the asymmetric version of KZG below.

Construction 8 (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 $\tau \leftarrow\!\!\$\ \mathbb{Z}_p$ and output* $\mathsf{crs} := \{g_1, g_1^{\tau}, g_1^{\tau^2}, \ldots, g_1^{\tau^d}, g_2, g_2^{\tau}\}$.
- $\underline{\mathsf{com}_f \leftarrow \mathsf{Com}(\mathsf{crs}, f(X)) :}$ *Let $f(X) = a_0 + a_1 X + \cdots + a_d X^d \in \mathbb{Z}_p[X]$. Use* crs *to compute and output $g_1^{f(\tau)} = g_1^{a_0} \cdot (g_1^{\tau})^{a_1} \ldots (g_1^{\tau^d})^{a_d} = g_1^{a_0 + a_1\tau + \cdots + a_d\tau^d} \in \mathbb{G}_1$.*
- $\underline{(f(i), \pi_i) \leftarrow \mathsf{Open}(\mathsf{crs}, f(X), i) :}$ *To open $f(X)$ at i, let $q_i(X) := \frac{f(X) - f(i)}{X - i} \in \mathbb{Z}_p[X]$*[6]*. 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 $\{g_1, g_1^{\tau}, \ldots, g_1^{\tau^d}, g_2, g_2^{\tau}\}$, 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 $\mathbb{G}_1$.

2.9 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 9 (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 $r \leftarrow\!\!\$\ \mathbb{Z}_p$ 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* $\mathsf{decom} = (x, r)$.

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

- $\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{com}, x, r)$: *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.

$\mathsf{Prove}(\mathsf{com}_{\mathsf{ped}}; (v, r)) \rightarrow \pi_{\mathsf{ped}}$: 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 $s_1, s_2 \leftarrow\!\$\ \mathbb{Z}_p$ and sends $a := g_1^{s_1} h_1^{s_2}$ to the verifier.
2. The verifier sends back a uniform challenge $c \leftarrow\!\$\ \mathbb{Z}_p$.
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 $\pi_{\mathsf{ped}} := (a, c, (t_1, t_2))$.

$\mathsf{Vrfy}(\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}) \rightarrow \{0,1\}$: 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].

2.10 Leftover Hash Lemma

We use the presentation of the leftover hash lemma (LHL) [IZ89] from [AMPR19].[7] 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].

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

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*

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

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.

2.11 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 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 $\text{REAL}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)$ 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 $\{\text{REAL}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$. 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 $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)$ and corresponding distribution ensemble $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$.

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 7. *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].)

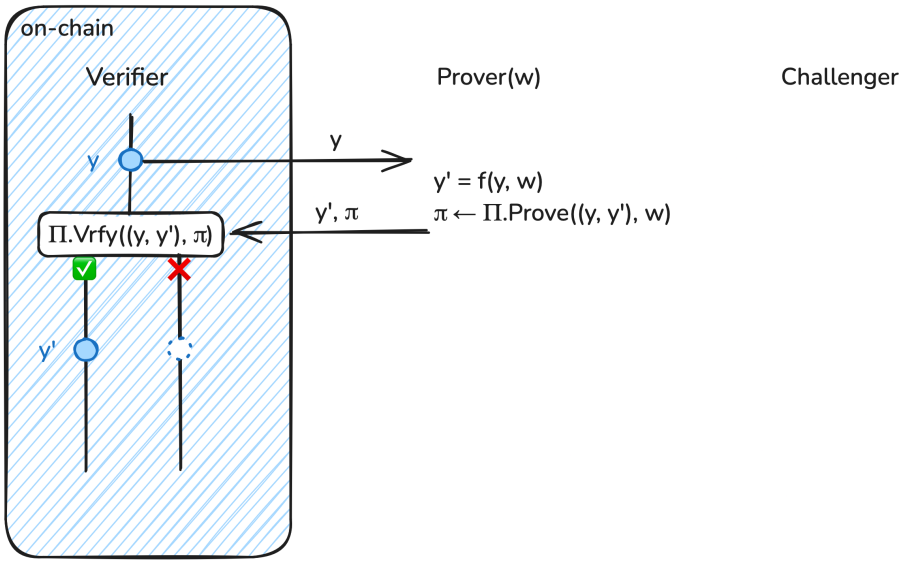

Figure 2: **Using VC to move computation off-chain.** The off-chain "prover" applies the function f to input y and potentially an auxiliary off-chain input w to get the result $y' = f(y, w)$. (In the case of a zk-rollup, f is the state transition function, y is the previous on-chain state, w is a batch of transactions, and y' is the new state after applying the transactions in w to y.) It posts y' and a proof π of its correctness, which is verified on-chain before the output y' is accepted. This paradigm does not require any challengers.

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 (VC)* [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).

VC leads to a paradigm in which smart contracts, while capable of arbitrary computation, primarily act as verifiers and outsource all significant computation off-chain (see Figure 2). A motivating application is so-called "zk"-

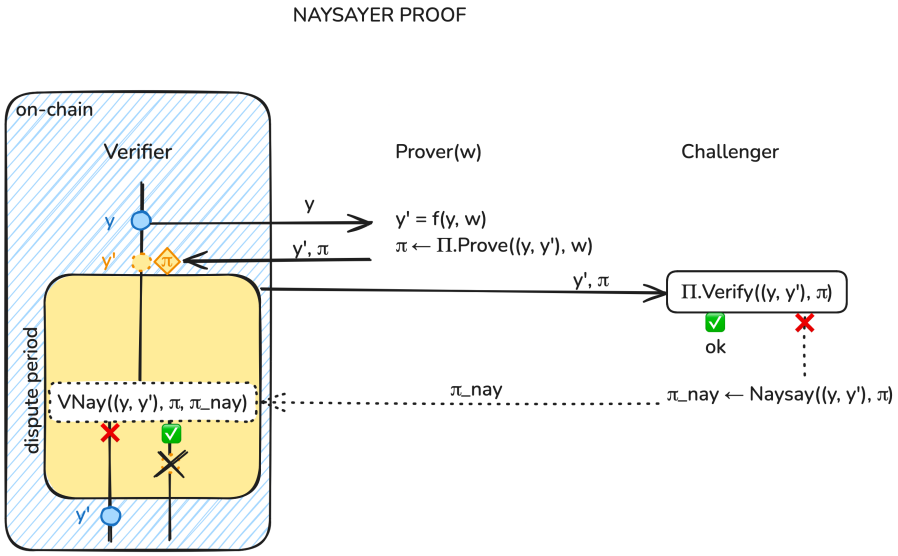

Figure 3: **The naysayer proof approach.** As in VC, the off-chain prover computes $y' = f(y, w)$ and π, which it posts on-chain. This time, the proof is *not* verified on-chain, but is provisionally accepted while waiting for the challenge period to pass. Any party can verify π off-chain and, if it fails, issue a challenge by creating a naysayer proof π_{nay}. The on-chain verifier checks any submitted naysayer proofs, and if they pass, it rejects the claimed result y'. If the challenge period elapses without any successful naysaying, y' is accepted.

rollups[8] [sta, zks, azt, dyd, scr], which combine 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.[9]

We observe that this proof verification is often wasteful. In most applications, provers have strong incentives to post only 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.[10]

This state of affairs motivates us to propose a new paradigm called *naysayer proofs* (Figure 3). In this paradigm, the verifier (e.g., a rollup smart contract) optimistically accepts a submitted proof without verifying its correctness. In-

[8] Note that the "zk" part of the name is often a misnomer, since these services do not necessarily offer the zero-knowledge property, and in fact they often do not.

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

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

stead, 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 successfully naysays the original proof before the end of the challenge 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 (the same is true of fraud proofs; see Section 3.1). We expect this to almost always be the case in practice. Second, even in the pessimistic case, we will see below that checking the naysayer proof can be much more efficient than checking the original proof. 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; see Section 3.3).

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 on-chain.

We will formally define naysayer proofs in Section 3.3 and show that every succinct proof system has a logarithmic-size and constant-time naysayer proof. First, we discuss related work in Section 3.1 and define our system model in more detail in Section 3.2. In Section 3.4, we construct naysayer proofs for four concrete proof systems and evaluate their succinctness. We discuss storage considerations in Section 3.5 and conclude with some open directions in Section 3.6.

3.1 Related Work

A concept related to the naysayer paradigm is *refereed delegation* [FK97]. The idea has found widespread adoption [TR19, KGC+18] under the name "fraud proofs" or "fault proofs" and is the core idea behind *optimistic rollups* [Eth23b, Lab23, Opt23a]. In classic refereed delegation, a server can output a heavy computation to two external parties, who independently compute and return the result. If the reported results disagree, the parties engage in a *bisection protocol* which pinpoints the single step of the computation which gave rise to the disagreement by recursively halving the computational trace (essentially performing a binary search). Once the discrepancy has been reduced to a single step of the computation, the original server can re-execute only that step to determine which of the two parties' results is correct.

In the context of optimistic rollups, a "prover" performs the computation off-chain and posts the result on-chain, where it is provisionally accepted. Any

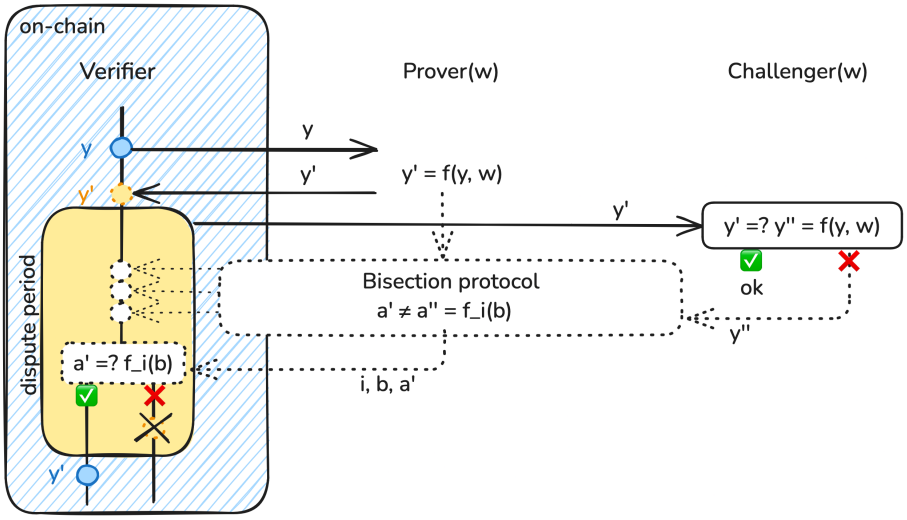

Figure 4: **Interactive fraud "proofs".** Like naysayer proofs, fraud "proofs" make use of challengers and challenge periods. Again, the off-chain "prover" computes $y' = f(y, w)$, but without providing any proof of correctness π. During the challenge period, anyone (with access to w, e.g., the batch of transactions) can re-compute $f(y, w)$. If the result y'' does not equal y', the party engages in a bisection protocol with the original "prover" to narrow the disagreement to a single step of the computation $f_i(b)$. The defender and challenger submit their respective one-step results $a' \neq a''$ to the on-chain verifier, who re-executes $f_i(b)$ on-chain. Based on the result, it may reject the original claim y'. If the challenge period elapses without any successful fraud proofs, y' is accepted.

party can then challenge the correctness of the result by posting a challenge on-chain and engaging in the bisection protocol with the prover via on-chain messages (Figure 4). (The term "fraud proof" or "fault proof" refers to these messages.[11]) Once the problematic step is identified, it is re-executed on-chain to resolve the dispute. A dispute can also be resolved *non-interactively* by re-running the entire computation on-chain in the event of a dispute (Figure 5), an approach initially taken by Optimism [Sin22, Buc]. If no one challenges the prover's result before the end of the *challenge period* (typically 7 days [Fic]), it is accepted and irreversibly commited on the layer-1 chain.

The naysayer approach offers significant speedups for the challenger over fraud proofs, since for succinct proof systems, verification is much more efficient than the original computation. Notice that there is a slight semantic difference between fraud proofs and naysayer proofs: A fraud proof challenges the correctness of the prover's *computation*, and thus can definitively show that the

[11] Despite the name, this is not actually a proof system, nor does it depend on any proof system.

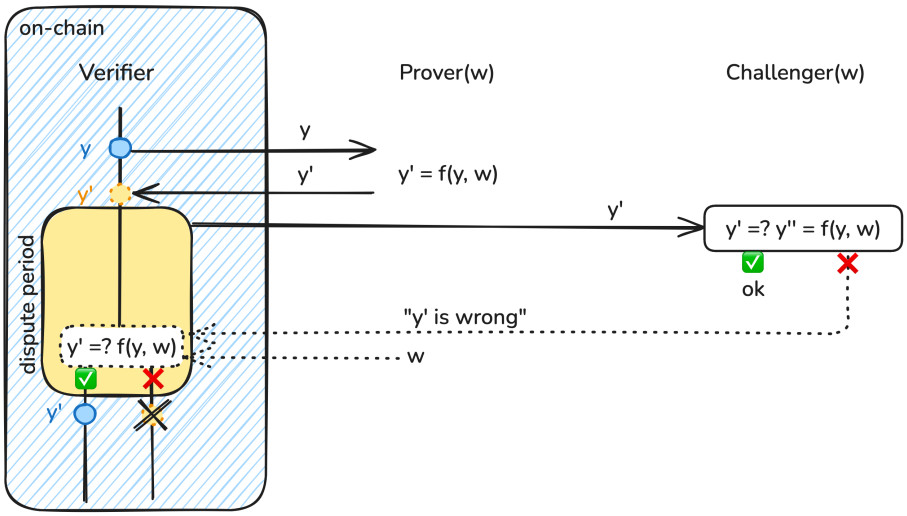

Figure 5: **Non-interactive fraud "proofs".** The non-interactive version functions similarly, except that there is no bisection game. Instead, the on-chain verifier simply re-executes the entire computation $f(y, w)$ on-chain to decide whether or not to reject y'. Again, if the challenge period elapses without any challenges, y' is accepted.

computation output is incorrect. In contrast, a naysayer proof challenges the correctness of the *accompanying proof*, and can therefore only show that the proof is invalid—the computation itself may still have been correct. In practice, there is no reason for this distinction to come up: a prover who performs the computation honestly has no incentive to attach an incorrect proof since that would mean it wasted computational power to compute y' for nothing.

Comparison. We compare classic verifiable computation, fraud proofs, and our new approach in Table 1.

oth fraud proofs and naysayer proofs work under an optimistic assumption, meaning a computation is accepted as correct unless some party challenges it. This requires assuming that at least one honest party is present to challenge any incorrect results. This party must also be *able* to submit a challenge, meaning we rely on the censorship-resistance of the underlying blockchain and assume new messages are added within some known, bounded delay. VC does not rely on these assumptions since every computation is checked at the time the result is submitted. It is well known that this leads to the faster on-chain finality of zk-rollups, which use the VC paradigm and thus do not require a challenge period.

Except for the interactive version of fraud proofs, all of the approaches require only a single message from the (off-chain) prover or challenger to the (on-

	VC	fraud proof (interactive)	fraud proof (non-interactive)	naysayer proof
No optimistic assumption	●	○	○	○
Non-interactive	●	○	●	●
Off-chain f	●	●	◐	●
Off-chain Π.Vrfy	○	-	-	●
Witness-independent challenge	●	○	○	●
Witness-independent resolution	●	◐	○	●
No Π.Prove	○	●	●	○

Table 1: Trade-offs between VC, fraud proofs, and naysayer proofs.

chain) verifier. Optimistic rollups almost universally employ interactive fraud proofs, requiring multiple on-chain messages in case of a dispute. This means the challenge period must be long enough to ensure that all the messages of the bisection protocol can be posted on-chain, even in the presence of some fraction of malicious consensus nodes who delay inclusion of the challenger's (or prover's) messages. We conjecture that by virtue of having a non-interactive challenge phase, naysayer proofs (and non-interactive fraud proofs) admit a shorter challenge period. Furthermore, the challenge period must also be long enough to accommodate the challenge resolution protocol to run on-chain. Thus, naysayer proofs should have an advantage even over non-interactive fraud proofs, since for all practical use cases, the on-chain resolution of the former (verifying a naysayer proof) will always be faster than re-computing the function f on-chain.

As is their goal, none of the approaches require running the original computation f on-chain, except for non-interactive fraud proofs in the (rare) case of a dispute. Compared to VC, fraud proofs and naysayer proofs do not require running proof verification on-chain (fraud proofs do not use a proof system at all). However, fraud proofs require the full computation input (including any off-chain input w, which we refer to as the witness) to be available to potential challengers and at least in part to the verifier. Neither VC nor naysayer proofs require this information to verify the correctness of the output y': they use only the statement and proof, which are already available on-chain.

Finally, a major advantage of fraud proofs is that they do not use any proof system at all. This makes them much easier to implement and deploy. VC and naysayer proofs, on the other hand, require computing a succinct proof, which is costly both in terms of implementation complexity and prover runtime. However, the design and efficiency of the bisection protocol can depend significantly on the programming model used [KGC+18] and the particular function f being computed [PD16, PB17, SNBB19, SJSW19]. We thus 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.

3.2 Model

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). Fix a function $f : \mathcal{X} \times \mathcal{W} \to \mathcal{Y}$ and let $\mathcal{L}_f$ be the language $\{(x, y) : \exists w \text{ s.t. } y = f(x, w)\}$. Let $\mathcal{R}_f = \{((x, y), w)\}$ be the corresponding relation. We assume f, x are known by all parties. When $f : \mathcal{Y} \times \mathcal{W} \to \mathcal{Y}$ is a state transition function with $y' = f(y, w)$, this corresponds to the rollup scenarios described above.

Prover The prover posts y and a proof π to the bulletin board claiming $(x, y) \in \mathcal{L}_f$.

Verifier The verifier does not directly verify the validity of y or π, rather, it waits for time T_{nay}. If no one naysays (y, π) within that time, the verifier accepts y. 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 y.

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

Note that, due to the optimistic paradigm, we must assume a synchronous communication model: in partial synchrony or asynchrony, the adversary can arbitrarily delay the posting of naysayer proofs, and one cannot enforce soundness of the underlying proofs. Furthermore, we assume that the public bulletin board offers censorship-resistance, i.e., anyone who wishes to write to it can do so successfully within a certain time bound. Finally, we assume that there is *at least one honest party* who will submit a naysayer proof for any invalid π.

3.3 Formal Definitions

Next, we introduce a formal definition and syntax for naysayer proofs. A naysayer proof system Π_{nay} can be seen as a "wrapper" around an underlying proof system Π. For example, Π_{nay} defines a proving algorithm $\Pi_{\mathsf{nay}}.\mathsf{Prove}$ which uses the original prover $\Pi.\mathsf{Prove}$ as a subroutine.

Definition 8 (Naysayer proof)**.** *Given a non-interactive proof system* $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vrfy})$ *for a NP language* $\mathcal{L}$*, the naysayer proof system corresponding to* Π *is a tuple of PPT algorithms* $\Pi_{\mathsf{nay}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Naysay}, \mathsf{VrfyNay})$ *defined as follows:*

$\mathsf{Setup}(1^\lambda, 1^{\lambda_{\mathsf{nay}}}) \xrightarrow{\$} (\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}})$**:** *Given (potentially different) security parameters* 1^λ *and* $1^{\lambda_{\mathsf{nay}}}$*, output two common reference strings* crs *and* $\mathsf{crs}_{\mathsf{nay}}$*. This algorithm may use private randomness.*

$\mathsf{Prove}(\mathsf{crs}, x, w) \to \pi'$**:** *Given a statement* x *and witness* w *such that* $(x, w) \in \mathcal{R}$*, output* $\pi' = (\pi, \mathsf{aux})$*, where* $\pi \leftarrow \Pi.\mathsf{Prove}(\mathsf{crs}, x, w)$*.*

$\mathsf{Naysay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \mathsf{td}_{\mathsf{nay}}) \to \pi_{\mathsf{nay}}$**:** *Given a statement x and values $\pi' = (\pi, \mathsf{aux})$ where π is a (potentially invalid) proof that $x \in \mathcal{L}$ using the proof system Π, output a naysayer proof π_{nay} disputing π. This algorithm may also make use of some (private) trapdoor information $\mathsf{td}_{\mathsf{nay}}$.*

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

A trivial naysayer proof system always exists in which $\pi_{\mathsf{nay}} = \top$, $\pi' = (\pi, \bot)$, and $\mathsf{VrfyNay}$ simply runs the original verification procedure, outputting 0 if $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 0$ and $\bot$ otherwise. 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 $\mathsf{td}_{\mathsf{nay}} = \bot$, we say Π_{nay} is a *public* naysayer proof (see Section 3.4.4 for an example of a non-public naysayer proof).

Definition 9 (Naysayer completeness)**.** *Given a proof system Π, a naysayer proof system $\Pi_{\mathsf{nay}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Naysay}, \mathsf{VrfyNay})$ is* complete *if, for all honestly generated $\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}$ and all values of aux,[12] given an invalid statement-proof pair (x, π), Naysay outputs a valid naysayer proof π_{nay}. That is, for all $\lambda, \lambda_{\mathsf{nay}} \in \mathbb{N}$ and all aux, x, π,*

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

Definition 10 (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 honestly generated $\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}$, all $(x, w) \in \mathcal{R}_{\mathcal{L}}$, and all correct proofs π', $\mathcal{A}$ produces a passing naysayer proof π_{nay} with at most negligible probability. That is, for all $\lambda, \lambda_{\mathsf{nay}} \in \mathbb{N}$, and all $\mathsf{td}_{\mathsf{nay}}$,*

$$\Pr\left[\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \pi_{\mathsf{nay}}) = 0 \;\middle|\; \begin{array}{c} (\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}) \leftarrow \mathsf{Setup}(1^{\lambda}, 1^{\lambda_{\mathsf{nay}}}) \wedge \\ (x, w) \in \mathcal{R}_{\mathcal{L}} \;\wedge \pi' \leftarrow \mathsf{Prove}(\mathsf{crs}, x, w) \wedge \\ \pi_{\mathsf{nay}} \leftarrow \mathcal{A}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \mathsf{td}_{\mathsf{nay}}) \end{array}\right] \leq \mathsf{negl}(\lambda_{\mathsf{nay}}).[13]$$

Next, we show that every proof system has corresponding naysayer proof system with a logarithmic-sized (in the size of the verification circuit) naysayer proofs and constant verification time (i.e., a succinct naysayer proof system).

[12] We do not place any requirement on aux.

[13] If we assume aux is computed correctly, the second line of the precondition can be simplified to see that Π_{nay} is a sound proof system for the language $\mathcal{L}_{\mathsf{nay}} = \{(x, \pi) : x \notin \mathcal{L} \vee \Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi) \neq 1\}$.

Lemma 2. *A claimed satisfying assignment for a circuit $C : \mathcal{X} \to \{0,1\}$ on input $x \in \mathcal{X}$ is efficiently naysayble. That is, if $C(x) \neq 1$, there is an $O(\log |C|)$-size proof of this fact which can be checked in constant-time, assuming oracle access to the wire assignments of $C(x)$.*

Proof. Without loss of generality, let C be a circuit of fan-in 2.

If $C(x) \neq 1$, then there must be some gate of C for which the wire assignment is inconsistent. Let i be the index of this gate (note $|i| \in O(\log |C|)$). To confirm that $C(x) \neq 1$, a party can re-evaluate the indicated gate G_i on its inputs a, b and compare the result to the output wire c. That is, if $G_i(a,b) \neq c$, the verifier rejects the satisfying assignment. □

Theorem 1. *Every proof system Π with $\mathsf{poly}(x)$ verification complexity has a succinct naysayer proof.*

Proof. Given any proof system Π, the evaluation of $\Pi.\mathsf{Vrfy}(\mathsf{crs}, \cdot, \cdot)$ can be represented as a circuit C. (We assume this circuit description is public.) Then the following is a complete and sound naysayer proof system Π_{nay}:

$\mathsf{Setup}(1^\lambda, 1^{\lambda_{\mathsf{nay}}})$**:** Output $\mathsf{crs} \leftarrow \Pi.\mathsf{Setup}(1^\lambda)$ and $\mathsf{crs}_{\mathsf{nay}} := \varnothing$.

$\mathsf{Prove}(\mathsf{crs}, x, w) \to \pi'$**:** Let $\pi \leftarrow \Pi.\mathsf{Prove}(\mathsf{crs}, x, w)$ and aux be the wire assignments of $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi)$. Output $\pi' = (\pi, \mathsf{aux})$.

$\mathsf{Naysay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \mathsf{td}_{\mathsf{nay}})$**:** Parse $\pi' = (\pi, \mathsf{aux})$[14] and output $\pi_{\mathsf{nay}} := \top$ if $\mathsf{aux} = \mathsf{aux}' \| 0$. Otherwise, evaluate $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi)$. If the result is not 1, search aux to find an incorrect wire assignment for some gate $G_i \in C$. Output $\pi_{\mathsf{nay}} := i$.

$\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \pi_{\mathsf{nay}})$**:** Parse $\pi' = (\cdot, \mathsf{aux})$ and $\pi_{\mathsf{nay}} = i$. If $\mathsf{aux} = \mathsf{aux}' \| 0$, output 0, indicating rejection of the proof π'. Otherwise, obtain the values $\mathsf{in}, \mathsf{out} \in \mathsf{aux}$ corresponding to the gate G_i and check $G_i(\mathsf{in}) \stackrel{?}{=} \mathsf{out}$. If the equality does not hold, output $\perp$; else output 0.

Completeness (if a π fails to verify, we can naysay (π, aux)) follows by Lemma 2. If $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi) \neq 1$, then we have two cases: If aux is consistent with a correct evaluation of $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi)$, either $\mathsf{aux} = \mathsf{aux}' \| 0$ (and $\mathsf{VrfyNay}$ rejects) or we can apply the lemma to find an index i such that $G_i(\mathsf{in}) \neq \mathsf{out}$ for $\mathsf{in}, \mathsf{out} \in \mathsf{aux}$, where $G_i \in C$. Alternatively, if aux is not consistent with a correct evaluation, there must be some gate (with index i') which was evaluated incorrectly, i.e., $G_{i'}(\mathsf{in}) \neq \mathsf{out}$ for $\mathsf{in}, \mathsf{out} \in \mathsf{aux}$.

Soundness follows by the completeness of Π. If $(x, w) \in \mathcal{R}_{\mathcal{L}}$ and $\pi' = (\pi, \mathsf{aux})$ is computed correctly, completeness of Π implies $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x, \pi) = 1$. Since aux is correct, it follows that $\mathsf{aux} \neq \mathsf{aux} \| 0$ and $G_i(\mathsf{in}) = \mathsf{out}$ for all $i \in |C|$ and corresponding values $\mathsf{in}, \mathsf{out} \in \mathsf{aux}$. Thus there is no index i which will cause $\mathsf{VrfyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), i)$ to output 0.

[14] if $|\mathsf{aux}|$ is larger than the number of wires in C, truncate it to the appropriate length

Succinctness of π_{nay} follows from the fact that $|i| = \log|\Pi.\mathsf{Vrfy}(\mathsf{crs},\cdot,\cdot)| \in \mathcal{O}(\mathsf{polylog}(|x|))$. Furthermore, it is clear to see that the runtime of $\mathsf{VrfyNay}$ is constant. □

The proof of Theorem 1 gives a generic way to build a succinct naysayer proof system for any proof system Π with polynomial-time verification. For strongly succincttodo: define or change proof systems, the generic construction even allows efficient naysaying, since the runtime of Naysay depends only on the runtime of $\Pi.\mathsf{Vrfy}$, which is sublinear if Π is strongly succinct.

Notice that although the syntax gives $\pi' = (\pi, \mathsf{aux})$ as an input to the $\mathsf{VrfyNay}$ algorithm, in the generic construction the algorithm does not make use of π. Thus, if the naysayer rollup from Figure 3 were instantiated with this generic construction, π would not need to be posted on-chain since the on-chain verifier (running the $\mathsf{VrfyNay}$ algorithm) will not use this information. In fact, the verifier wouldn't even need most of aux—only the values corresponding to the gate G_i, which is determined by π_{nay}. Thus, although π' must be available to all potential naysayers, only a small (adaptive) fraction of it must be accessible on-chain. We describe in Section 3.5 how to leverage this insight to reduce the storage costs of a naysayer rollup. First, in Section 3.4, we show application-specific naysayer constructions which are more efficient than the generic naysayer proof.

3.4 Constructions

In Section 3.4.1, we show a concrete example of the generic naysayer construction from Theorem 1 applied to Merkle trees.

todo: moved here, edit: Many proof systems have some repetitive structure in their verification algorithm. This structure allows for more efficient naysaying. Examples include proofs with multi-round amplification, verification as a conjunction of repeated checks (e.g., multiple bilinear pairing checks [GWC19]), or recursive reduction (e.g., Pietrzak's proof of exponentiation [Pie19]).

We then highlight three naysayer proof constructions which take advantage of repetition in the verification algorithm to achieve better naysayer performance: the FRI polynomial commitment scheme (Section 3.4.2) and two post-quantum signature schemes (Section 3.4.3).

Then, in Section 3.4.4, we give an example of a non-public naysayer proof which uses a trapdoor to reduce the size and verification complexity of the naysayer proof.

Finally, in Section 5.7.2 we estimate the performance of each naysayer proof system compared to that of the original proof system.

3.4.1 Merkle Commitments

Merkle trees [Mer88] and their variants are ubiquitous in modern systems, including Ethereum's state storage [Eth24b]. A Merkle tree can be used to commit

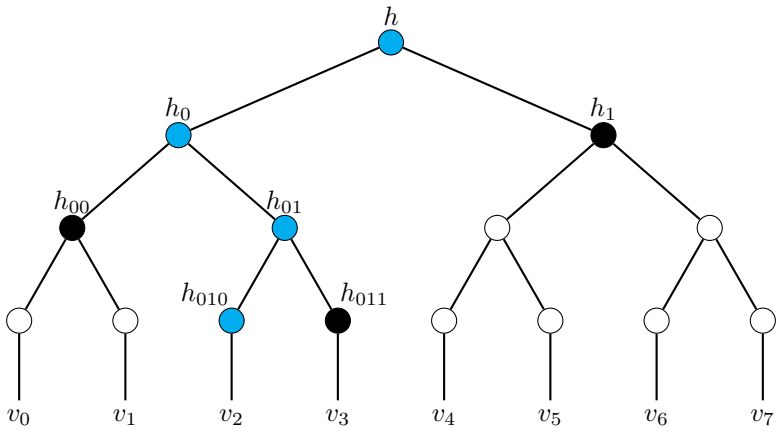

Figure 6: Each node in a Merkle tree consists of a hash of its children. The root h is a commitment to the vector of leaves $(v_0, v_1, \ldots, v_7)$. An opening proof for the element v_2 is its copath (black nodes); the "verification trace" for the proof is the path (blue nodes).

to a vector $\mathbf{v}$ of elements as shown in Figure 6, with the root h acting as a commitment to $\mathbf{v}$. The party who created the tree can prove the inclusion of some element v_i at position i in the tree by providing the corresponding copath.

For example, to open the leaf at position 2, a prover provides its value v_2 and an opening proof $\pi = (h_{011}, h_{00}, h_1)$ consisting of the copath from the leaf v_2 to the root h. The proof π is checked by using its contents to recompute the root h' starting with v_2, then checking that $h = h'$. This involves recomputing the nodes along the path from the leaf to the root (the blue nodes in the figure). These nodes can be seen as a "verification trace" for the proof π.

In the context of a naysayer proof system, the prover provides π along with the verfication trace $\mathsf{aux} = (h_{010}, h_{01}, h_0)$. A naysayer can point out an error at a particular point of the trace by submitting the incorrect index of aux (e.g., $\pi_{\mathsf{nay}} = 1$ to indicate h_{01}). The naysayer verifier checks π_{nay} by computing a single hash using π and oracle access to aux, e.g., checking $H(h_{010}, h_{011}) \stackrel{?}{=} h_{01}$, where $h_{010}, h_{01} \in \mathsf{aux}$ and $h_{011} \in \pi$. This is the generic construction from Theorem 1.

3.4.2 FRI

The Fast Reed-Solomon IOP of proximity (FRI) [BBHR18a] is used as a building block in many non-interactive proof systems, including the STARK IOP [BBHR18b]. Below, we describe only the parts of FRI as applied in STARK. We refer the reader to the cited works for details.

The FRI commitment to a polynomial $p(X) \in \mathbb{F}[X]^{\leq d}$ 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 a constant $0 < \rho < 1$ (the Reed-Solomon rate parameter). We focus on the verifier's cost in the proof of proximity. Let δ be a parameter of the scheme such that $\delta \in (0, 1 - \sqrt{\rho})$. The prover sends $\log d + 1$ values (roots of successive "foldings" of the original Merkle tree, plus the value of the constant polynomial encoded by the final tree). The verifier makes $q = \lambda / \log(1/(1-\delta))$ queries to ensure $2^{-\lambda}$ soundness error; the prover responds to each query with $2 \log d$ Merkle opening proofs (2 for each folded root). For each query, the verifier must check each Merkle authentication path, amounting to $\mathcal{O}\left(\log d \log \rho^{-1} d\right)$ hashes per query. Furthermore, it must perform $\log d$ arithmetic checks (roughly 3 additions, 2 divisions, and 2 multiplications in $\mathbb{F}$ per folding) per query to ensure the consistency of the folded evaluations. Therefore, the overall FRI verification consists of $\mathcal{O}\left(\lambda \log^2 d\right)$ hashes and $\mathcal{O}\left(\lambda \log d\right)$ field operations.

A FRI proof is invalid if any of the above checks fails. Therefore a straightforward naysayer proof $\pi^{\mathsf{FRI}}_{\mathsf{nay}} = (i, j, k)$ need only point out a single Merkle proof (the jth proof for the ith query, $i \in [q], j \in [2 \log d]$) or a single arithmetic check $k \in [q \log d]$ which fails. The naysayer verifier only needs to recompute that particular check: $\mathcal{O}\left(\log \rho^{-1} d\right)$ hashes in the former case[15] or a few arithmetic operations over $\mathbb{F}$ in the latter.

3.4.3 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]. Compared to their classical counterparts, post-quantum signatures generally have either substantially larger signatures or substantially larger public keys.[16] Since this makes post-quantum signatures expensive to verify on-chain, these schemes are prime candidates for the naysayer proof paradigm.

CRYSTALS-Dilithium [DKL+18]. We give a simplified version of signature verification in lattice-based signatures like CRYSTALS-Dilithium. In these schemes, the verifier checks that the following holds for a signature $\sigma = (\mathbf{z}_1, \mathbf{z}_2, c)$, public key $\mathsf{pk} = (\mathbf{A}, \mathbf{t})$, and message M:

$$\|\mathbf{z}_1\|_\infty < \beta \wedge \|\mathbf{z}_2\|_\infty < \beta \wedge c = H(M, \mathbf{w}, \mathsf{pk}). \tag{2}$$

Here β is a constant, $\mathbf{A} \in R_q^{k \times \ell}$, $\mathbf{z}_1 \in R_q^\ell$, $\mathbf{z}_2, \mathbf{t} \in R_q^k$ for the polynomial ring $R_q := \mathbb{Z}_q[X]/(X^d + 1)$, and $\mathbf{w} = \mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t} \mod q$. (Dilithium uses

[15] One could use a Merkle naysayer proof (Section 3.4.1) to further reduce the naysayer verification from checking a full Merkle path to a single hash evaluation.

[16] Considering the NIST-standardized post-quantum signature schemes, Dilithium has 1.3KB public keys and 2.4KB signatures for its lowest provided security level (NIST level 2) [DKL+21]; the "small" variant of SPHINCS+ for NIST level 1 has 32B public keys but 7.8KB signatures [ABB+22]; and FALCON at level 1 has 897B public keys and 666B signatures [FHK+20]. By comparison, 2048-bit RSA requires only 256B both for public keys and signatures while offering comparable security [Gir20] (only against classical adversaries, of course).

$d = 256$.) We will write elements of R_q as polynomials $p(X) = \sum_{j \in [d]} \alpha_j X^j$ with coefficients $\alpha_j \in \mathbb{Z}_q$. Since Equation (2) is a conjunction, the naysayer prover must show that

$$(\exists z_i \in \mathbf{z}_1, \mathbf{z}_2 : \|z_i\|_\infty > \beta) \vee c \neq H(M, \mathbf{w}, \mathsf{pk}). \tag{3}$$

If the first check of Equation (2) fails, the naysayer gives an index i for which the infinity norm of one of the polynomials in $\mathbf{z}_1$ or $\mathbf{z}_2$ is large. (In particular, it can give a tuple (b, i, j) such that $\alpha_j > \beta$ for $z_i = \cdots + \alpha_j X^j + \cdots \in \mathbf{z}_b$.)[17]

If the second check fails, the naysayer indicates that clause to the naysayer verifier, who must recompute $\mathbf{w}$ and perform a single hash evaluation which is compared to c.

Overall, π_{nay} is a tuple (a, b, i, j) indicating a clause $a \in [2]$ of Equation (3), the vector $\mathbf{z}_b$ with $b \in [2]$, an entry $i \in [\max\{k, \ell\}]$ in that vector, and the index $j \in [d]$ of the offending coefficient in that entry. Since $k \geq \ell$, we have $|\pi_{\mathsf{nay}}| = (2 + \log k + \log d)$ bits. The verifier is very efficient when naysaying the first clause, and only slightly faster than the original verifier for the second clause.

SPHINCS+ [BHK+19]. The signature verifier in SPHINCS+ checks several Merkle authentication proofs, requiring hundreds or even thousands of hash evaluations. An efficient naysayer proof can be easily devised akin to the Merkle naysayer described in Section 3.4.1. Given a verification trace, the naysayer prover simply points to the hash evaluation in one of the Merkle-trees where the signature verification fails.

3.4.4 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 particularly efficient naysayer proof if the naysayer knows at least one of the shuffled elements. Let us consider the simple case of shuffling public keys. The shuffler wishes to prove membership in the following NP language:

$$\begin{aligned}\mathcal{L}_{perm} := \{((\mathsf{pk}_i, \mathsf{pk}'_i)_{i=1}^n, R) : \exists r, w_1, \ldots, w_n \in \mathbb{F}_p, \sigma \in \mathsf{Perm}(n) \\ \text{s.t. } \forall i \in [n], \mathsf{pk}_i = g^{w_i} \wedge \mathsf{pk}'_i = g^{r \cdot w_{\sigma(i)}} \wedge R = g^r\}.\end{aligned}$$

Here $\mathsf{Perm}(n)$ is the set of all permutations $f : [n] \to [n]$.

[17] The same idea can be applied to constructions bounding the ℓ_2 norm, but with lower efficiency gains for the naysayer verifier, who must recompute the full ℓ_2 norm of either $\mathbf{z}_1, \mathbf{z}_2$.

	Merkle opening	FRI opening	CRYSTALS-D.	Shuffle proof
$\lvert\pi\rvert$	$\log n\ \mathbb{H}$	$\mathcal{O}\left(\lambda \log^2 d\right)\mathbb{H} + \mathcal{O}\left(\lambda \log d\right)\mathbb{F}$	$\mathcal{O}\left(\lambda\right)\mathbb{F}$	$\mathcal{O}\left(\sqrt{n}\right)\mathbb{G}$
Vrfy	$\log n\ \mathbb{H}$	$\mathcal{O}\left(\lambda \log^2 d\right)\mathbb{H} + \mathcal{O}\left(\lambda \log d\right)\mathbb{F}$	$\mathcal{O}\left(\lambda\right)\mathbb{F} + 1\mathbb{H}$	$\mathcal{O}\left(n\right)\mathbb{G}$
$\lvert\pi_{\mathsf{nay}}\rvert$	$\log\log n\ \mathbb{B}$	$2\log(q \log d) + 1\ \mathbb{B}$	$2 + \log k + \log d\ \mathbb{B}$	$\log n\ \mathbb{B} + 3\mathbb{G} + 1\mathbb{F}$
$\mathsf{VrfyNay}$ (best)	$1\mathbb{H}$	$\mathcal{O}\left(1\right)\mathbb{F}$	$\mathcal{O}(1)\mathbb{F}$	$\mathcal{O}\left(1\right)\mathbb{G} + 1\mathbb{H}$
$\mathsf{VrfyNay}$ (worst)	"	$\mathcal{O}\left(\log d\right)\mathbb{H}$	$\mathcal{O}\left(\lambda\right)\mathbb{F} + 1\mathbb{H}$	"

Table 2: Cost savings of the naysayer paradigm for the example applications in this section. $\mathbb{F}, \mathbb{G}, \mathbb{H}$ respectively denote either field elements/group elements/hash output sizes, or field/group/hash operations. We use $\mathbb{B}$ to indicate bits. In the case of naysayer proof sizes, we are able give exact numbers; therefore, we wish to point out that for Dilithium, $\lvert\pi_{\mathsf{nay}}\rvert \in \mathcal{O}\left(\log \lambda\right)$ since the parameter k depends on λ and d is a constant. todo: Can I get some estimated numbers here by looking at the implementations/papers of the underlying schemes, and then just the field/group element sizes in the relevant curves for the naysayer?

Suppose a party knows that for some $j \in [n]$, the prover did not correctly include $\mathsf{pk}'_j = g^{r \cdot w_j}$ in the shuffle. The party can naysay by showing that

$$(g, \mathsf{pk}_j, R, \mathsf{pk}'_j) \in \mathcal{L}_{DH} \wedge \mathsf{pk}'_j \notin (\mathsf{pk}_i, \cdot)_{i=1}^n$$

where $\mathcal{L}_{DH}$ is the language of Diffie-Hellman tuples[18]. To produce such a proof, however, the naysayer must know the discrete logarithm w_j. Unlike our previous examples, which were public naysayer proofs, this is an example of a private Naysay algorithm using $\mathsf{td}_{\mathsf{nay}} := w_j$. The naysayer proof is $\pi_{\mathsf{nay}} := (j, \mathsf{pk}'_j, \pi_{DH})$. The Diffie-Hellman proof can be checked in constant time and, with the right data structure for the permuted list (e.g., a hash table), so can the list non-membership. This, π_{nay} is a $\mathcal{O}\left(\log n\right)$-sized naysayer proof with $\mathcal{O}\left(1\right)$-verification, yielding in exponential savings compared to verifying the original Bayer-Groth shuffle proof.

3.4.5 Evaluation

We analyze the asymptotic cost savings for the verifiers in the four examples discussed in Sections 3.4.1 to 3.4.4 in Table 2. Note that the verifier speedup is exponential for verifiable shuffles and logarithmic for the FRI openings. For CRYSTALS-Dilithium, our naysayer proof is only weakly efficient (see Section 3.3) as there is no asymptotic gap in the complexity of the original signature verification and the naysayer verification in the worst case.

As for proof size, in all the examples, our naysayer proofs are logarithmically smaller than the original proofs. Furthermore, in most cases, the naysayer

[18] Membership in $\mathcal{L}_{DH}$ can be shown via a proof of knowledge of discrete logarithm equality [CP93] consisting of 2 group elements and 1 field element which can be verified with 4 exponentiations and 2 multiplications in the group.

proof consists of an *integer* index or indices rather that group or field elements; representing the former requires only a few bits compared to the latter (which are normally at least λ bits long). This means that, in practice, naysayer proofs can offer practically smaller proofs sizes even when the original proof is constant-size (e.g., a few group elements). todo: Need to check this with some common circuit sizes (e.g., the STARK verification circuit) and group element sizes (e.g., BN254 curve G1 element or other).

3.5 Storage Considerations

So far, we have assumed that the naysayer verifier can read the instance x, the original proof π and aux, and the naysayer proof π_{nay} entirely. A naysayer proof system thus requires increased storage (long-term for aux, and temporary for π_{nay} only in case of a challenge). However, the verifier only needs to compute $\mathsf{VrfyNay}$ instead of Vrfy. A useful naysayer proof system should therefore compensate for the increased storage by considerably reducing verification costs.

In either case, in blockchain contexts where storage is typically very costly, the approach of storing all data on chain may not be sufficient. Furthermore, as we pointed out previously, the verifier—the only entity which requires data to be stored on-chain in order to access it—does not access all of this data.

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. (Ethereum implemented this type of storage, commonly referred to as "blob data", using KZG commitments in EIP-4844 [BFL+22].)

This suggests an optimization for naysayer proofs in a blockchain context: the prover posts only a binding commitment $\mathsf{Com}(\pi')$, which the contract stores in S_{per}, while the actual proof $\pi' = (\pi, \mathsf{aux})$ 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 other case, when naysaying is necessary, the naysayer must send openings of the erroneous elements to the verifier (in S_{call}), who checks that these data elements are valid with respect to the on-chain commitment $\mathsf{Com}(\pi')$ stored in S_{per}. Note that most naysayer proof systems don't require reading all of π' for verification, so even the pessimistic case will offer significant savings over storing all of π' in S_{call}.

3.6 Extensions and Future Work

todo: edit this section

In the months since the publication of the paper [SGB24], naysayer proofs have already received interest from various groups of practitioners hoping to deploy them in production. Since this is a new paradigm, we see many areas to investigate both for immediate deployments and improving our understanding of this space and its implications:

Rollup design space. Naysayer proofs can be viewed as a way (one of several) of combining two established solutions—verifiable computation and fraud proofs—to offer different tradeoffs. There may be other ways to combine these two paradigms to achieve different tradeoffs which can be more suited to certain application scenarios.

todo: edit: Type-2 naysayer proofs are even more efficient in the optimistic case, as the prover *only sends the instance* x and no proofs at all, claiming without evidence that $(x, w) \in \mathcal{R}_{\mathcal{L}}$ (this is the same as in the fraud proof paradigm). On the other hand, if the prover's assertion is incorrect, i.e., $(x, w) \notin \mathcal{R}_{\mathcal{L}}$, then a naysayer prover provides the correct statement x' such that $(x', w) \in \mathcal{R}_{\mathcal{L}}$ and a corresponding "regular" proof π such that $\mathsf{Vrfy}(\mathsf{crs}, x', \pi) = 1$. Importantly, x' must correspond to the same witness w (which must be known to the challenger). 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. This can be seen as a naysayer proof system where $\pi' = \varnothing$, $\mathsf{aux}_{\mathsf{nay}} = w$, $\pi_{\mathsf{nay}} = (x', \pi_{x'})$, and $\mathsf{VrfyNay}$ runs $\Pi.\mathsf{Vrfy}(\mathsf{crs}, x', \pi_{x'})$. 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).

Naysaying zkVMs. An emerging trend in zk-rollups are so-called "zero-knowledge virtual machines", or zkVMs (again, the "zk" part may or may not hold). Noemi: This kinda creates an aux already (not for verification, but for the original computation); basically rephrases the statement as a conjunction of state transitions.

Other naysayer constructions. In our application-specific naysayer constructions, we looked at naysaying proof systems where verification has some repetitive structure. Are there other proof systems which have particularly efficient or useful naysayer proofs?

Computing aux. A drawback of (generic) naysayer proofs is that the prover must additionally run $\Pi.\mathsf{Vrfy}$ to create the verification trace aux. Is there a way to separate/outsource this to another party to reduce the added computational burden on a party already spending so much power to compute the proof?

Approach	NI	No TTPs	Efficient	Tally privacy	Ev. ballot privacy
Commit-reveal [FOO93, GY19]	○	●	●	●	○
Zero-knowledge proofs [PE23]	○	○	●	●	●
Fully homomorphic TLPs [MT19]	●	●	○	●	○
FHE [Gen09, CGGI16, dLNS17]	●	○	○	●	●
Multi-party computation [BDJ+06, AOZZ15]	○	◐	●	●	●
TLPs + homomorphic encryption [CJSS21]	●	◐	●	●	○
HTLPs (our approach)	●	●*	●	●	◐

* when using class groups

Table 3: Qualitative comparison of major cryptographic approaches for designing private auction/voting schemes. NI = non-interactive, Ev. = everlasting. [AOZZ15, CJSS21] require a trusted setup but no TTP. Everlasting ballot privacy can be added to our approach via an extension (Section 4.10).

Implementing query access. As mentioned in Section 3.5, a straightforward way to implement efficient and secure query access to π' is to assume some secure data availability service (e.g., Ethereum blob data). Are there other approaches to realizing this access trustlessly?

4 On-chain Voting and Auctions

(Parts of this section are taken/adapted from [GSZB24].)

Special Notation We will use n as the number of users, m as the number of candidates, and w as the maximum weight to be allocated to any one candidate in a ballot/bid ($n, m, w \in \mathbb{N}$). For simplicity and without loss of generality, we assume the user identities are unique integers $i \in [n]$. We generally use $i \in [n]$ to index users and $j \in [m]$ for candidates.

Auctions and voting are essential applications of Web3. For example, decentralized marketplaces run auctions to sell digital goods like non-fungible tokens (NFTs) [Ope23] or domain names [XWY+21], while decentralized autonomous organizations (DAOs) deploy voting schemes to enact decentralized governance [Opt23b]. Most auction or voting schemes currently deployed on blockchains, e.g., NFT auctions on OpenSea or Uniswap governance [FMW22], lack bid/ballot privacy. This can negatively influence user behavior, for example, by vote herding or discouraging participation [EL04, GY19, SY03]. The lack of privacy can cause surges in congestion and transaction fees as users try to outbid each other to participate, a negative externality for the entire network.

Existing private voting protocols [PE23, GG22, Pria] achieve privacy at the cost of introducing a trusted authority who is still able to view all submissions. Alternatively, the only private *and* trustless auction in deployment we are aware of [XWY+21] uses a two-round commit-reveal protocol: in the first

round, every party commits to their bid, and in the second round they open the commitments and the winner can be determined. Other protocols relying on more heavyweight cryptographic building blocks have been proposed in the literature. We summarize the various approaches for private voting and auctions in Table 3. Unfortunately, all of them suffer from at least one of the following limitations, hindering widespread adoption:

Interactivity. Interactivity is a usability hurdle that often causes friction in the protocols' execution. Mandatory bid/ballot reveals are also a target for censorship. A malicious party can bribe the block proposers to exclude certain bids or ballots until the auction/voting ends [PRF23].

Trusted third party (TTP). Many protocols [PE23] use a trusted coordinator to tally submissions during the voting/bidding phase. This introduces a strong assumption which is at odds with the trustless ethos of the blockchain ecosystem.

Inefficiency. Compute and storage costs are substantial bottlenecks in decentralized applications running on a public blockchain. Some approaches [CGGI16, dLNS17] avoid the previous pitfalls by relying on complex cryptographic primitives such as fully-homomorphic encryption (FHE), whose overheads are impractical in the blockchain setting.

4.1 Related Work

The cryptographic literature on both voting schemes and sealed-bid auctions is enormous, dating to the 1990s. However, most of these schemes are unsuitable for a fully decentralized and trust-minimized setting due to their inefficiency or reliance on trusted parties, i.e., tally authorities, servers running the public bulletin board, auctioneers, etc. Below, we review auction and voting protocols that use a blockchain as the public bulletin board.

Voting. The study of voting schemes for blockchain applications dates to at least 2017, when McCorry et al. [**?**] proposed a "boardroom" voting protocol for DAO governance. The main disadvantage of their protocol is that the entire protocol can be aborted due to a single party. Groth [**?**] and Boneh et al. [**?**] develop techniques to create ballot correctness proofs for various voting schemes. These protocols all have proofs with size linear in the number of candidates. We break this barrier by applying polynomial commitments and assuming a transparent, lightweight pre-processing phase. Applying HTLPs to voting was suggested when they were proposed by Malavolta and Thyagarajan [**?**]. However, they left the details of making such a protocol practical, secure, and efficient to future work. We aim to fill this gap with our techniques for various election types and our EVM implementation.

Auctions. Auctions are a natural fit for blockchains and were suggested as early as 2018 [**?**], albeit with a *trusted auctioneer*. Bag et al. introduced SEAL, a privacy-preserving sealed-bid auction scheme without auctioneers [**?**]. However, their protocol employs two rounds of communication since they apply the Hao-Zielinski Anonymous Veto network protocol [**?**]. Tyagi et al. proposed Riggs [**?**], a fair non-interactive auction scheme using timed commitments [**?**, §6]. This is perhaps the closest work to ours in implementing auctions (but not voting) in a fully decentralized setting using time-based cryptography. However, their design does not utilize homomorphism to combine puzzles, and as a result the gas costs are high. To achieve practicality Riggs relies on an optimistic second round in which users voluntarily open their puzzles. Chvojka et al. suggest a TLP-based protocol for both e-voting and auctions [**?**]. Their protocol has a per-auction trusted setup. In Section 4.10.5, we propose using HTLPs for distributed setup to reduce the trust assumption, which may be of independent interest.

Time-based cryptography. Time-based cryptography, which uses inherently sequential functions to delay the revelation of information, also has a lengthy history dating to Rivest, Shamir, and Wagner's 1996 proposal of time-lock encryption [**?**]. Numerous variants have emerged since then, including timed commitments [**?**], proofs-of-sequential-work [**?**], VDFs [**?**], and homomorphic time-lock puzzles [**?**], which we employ here. For a recent survey, we refer the reader to Medley et al. [**?**]. The only practical work we know of taking advantage of HTLPs is Bicorn [**?**], which builds a distributed randomness beacon with a single aggregate HTLP for an arbitrary number of entropy contributors.

Delay encryption constitutes the most recent development in time-based cryptography. Delay encryption allows time-lock encryption of a message to an identity by combining identity-based encryption with inherently sequential computation. This cryptographic primitive was introduced by Burdges and De Feo [**?**]. However, their isogeny-based construction has enormous memory requirements ($\approx$ 12 TiB), making their scheme impractical.

One can emulate time-based cryptography by applying stronger assumptions instead of assuming the sequentiality of repeated modular squaring. McFly [**?**] and tlock [**?**] build time-lock encryption from threshold trust, i.e., by assuming that a subset of signers intermittently and reliably releases a threshold signature on the current timestamp. Even though this is a stronger assumption, we view this as a promising alternative research direction. For instance, enabling the aggregation of multiple time-lock ciphertexts could make timelock encryption especially suitable for voting and auction applications; aggregation is not currently possible in the aforementioned schemes as they lack any homomorphism. Using any resulting homomorphic time-lock encryption schemes to build efficient voting and auction protocols is a further open problem. We leave these questions to future work.

4.2 Our Results

We aim to build voting and auction protocols that possess the following distinguishing features compared to prior work.

Trust-minimization. In our protocols, we do not want to assume (a quorum of) trusted third parties. The cryptographic voting literature extensively applies trusted parties, for example, to operate a public bulletin board or tally votes. The classic tools used in prior work, e.g. (fully) homomorphic encryption, inherently imply a handful of trusted parties to decrypt the ballots/bids. Apart from the liveness and safety of the blockchain consensus, we solely employ standard cryptographic assumptions.

One-round protocol. We argue that usability is one of the major challenges of deploying privacy-preserving voting and auction protocols in practice. Multi-round protocols, e.g., commit-reveal-style protocols, have incentive issues and considerable usability hurdles. We solve these pressing issues with efficient one-round protocols.

Ballot/bid privacy. Last but not least, we want to achieve ballot/bid privacy. Our approach naturally provides privacy until the end of the voting/bidding phase. This is sufficient for avoiding selective aborts and censorship, thereby ensuring fairness. Limited privacy can also be a desideratum in certain settings, e.g. in representative democracies where delegates' votes are published to encourage accountability. Alternatively, we show in Section 4.10.1 how we can add everlasting ballot/bid privacy to our protocol without sacrificing our two previously stated goals.

Coercion-resistance, i.e., the adversary's inability to coerce voters to cast specific ballots demanded by the adversary, is a crucial property of voting schemes. In the privacy-respecting e-voting literature, it is well-known that receipt-freeness (since the voters cannot prove how they voted) implies coercion-resistance [BT94]. However, we consider receipt-freeness as a non-goal in our protocol design. Still, we sketch an extension to our framework in Section 4.10.3, where we achieve coercion-resistance via a different pathway than receipt freeness. We leave it to future work to achieve the property of receipt freeness for on-chain voting schemes.

We introduce Cicada, a general framework for practical, privacy-preserving, and trust-minimized protocols for both auctions and voting. Cicada uses time-lock puzzles (TLPs) [RSW96] to achieve *privacy and non-interactivity* in a trustless and efficient manner. A TLP allows a party to "encrypt" a message to the future. Specifically, to recover the solution, one needs to perform a computation that is believed to be inherently sequential, with a parameterizable number of steps. Intuitively, the TLPs play the role of commitments to bids/ballots that any party can open after a predefined time, avoiding the reliance on a second *reveal* round.

Since solving a TLP is computationally intensive, ideally, an efficient protocol would require solving only a sublinear number of TLPs (in the number

of voters/bidders). Cicada achieves this via *homomorphic* TLPs (HTLPs): bids/ballots encoded as HTLPs can be "squashed" into a sublinear number of TLPs. Fully homomorphic TLPs are not practically efficient, but Malavolta and Thyagarajan [MT19] introduced efficient additively and multiplicatively homomorphic TLP constructions which we will describe below. This is clearly enough for simple constructions like first-past-the-post (FPTP) voting, but it remained an open problem how to apply HTLPs to realize more complicated auction and voting protocols, e.g., cumulative voting.

We show how to use HTLPs to build practically efficient, private, and non-interactive protocols for a special class of auction and voting schemes where the tallying procedure can be expressed as a linear function. We show that this limited class nonetheless includes many schemes of interest.

Moreover, we introduce a novel linear HTLP based on the exponential ElGamal cryptosystem [CGS97] over a group of unknown order. This construction is more efficient than the Paillier-based construction [MT19] for a small solution space $\mathcal{S} \subset \mathbb{Z}_N$, i.e., $\mathcal{S} = \{s : s \in \mathbb{J}_N \wedge \ s \ll N\}$. Here a puzzle Z is constructed as

$$(g^r, h^r y^s) \in (\mathbb{Z}_N^*)^2 \tag{4}$$

where $g, y \leftarrow\!\!\$\ \mathbb{Z}_N^*$ and again $h = g^{2^T}$. This scheme is only practical for small $\mathcal{S}$ since, in addition to recomputing h^r, recovering s requires brute-forcing the discrete modulus of y^s. We discuss the efficiency trade-off between these two constructions in Section 4.9.

Lifting the multiplicative HTLP of [MT19] (Construction 1) to put s in the exponent yields a more efficient linear HTLP for a small solution space $\mathcal{X} \subset \mathbb{Z}_N$, where $\mathcal{X} = \{s : s \in \mathbb{J}_N \wedge \ s \ll N\}$ (Figure 7, changes shown in blue). This can be viewed as a construction based on exponential ElGamal encryption over $\mathbb{Z}_N^*$.

Efficient vector encoding for HTLPs. In many voting schemes, a ballot consists of a vector indicating the voter's relative preferences or point allocations for all m candidates. To avoid solving many HTLPs, it is desirable to encode this vector into a single HTLP, which requires representing the vector as a single integer.

Note that as in the PNS approach, we set $M = nw+1$ to accommodate homomorphic addition of submissions; homomorphic multiplication, however, would require $M = w^n + 1$, and the primes in $\mathbf{p}$ would therefore be larger as well. Although the RNS has found application in error correction [KPT+22, TC14], side-channel resistance [PFPB18], and parallelization of arithmetic computations [AHK17, BDM06, GTN11, VNL+20], to our knowledge it has not been applied to voting schemes. We will show that RNS is in fact a natural fit for some voting schemes, in particular quadratic voting, where it results in more efficient proofs of ballot correctness.

Applied NIZKs in Groups of Unknown Order. We will use *non-interactive zero-knowledge* proofs (NIZKs) to enforce well-formedness of user submissions

Construction 10 (Efficient linear HTLP.)**.**

$\mathsf{HTLP.Setup}(1^\lambda, T) \xrightarrow{\$} \mathsf{pp}$**.** *Output* $\mathsf{pp} := (N, g, h, y)$*, where* $y \xleftarrow{\$} \mathbb{Z}_N^*$ *and the remaining parameters are the same as in constructions 1 and 2.*

$\mathsf{HTLP.Gen}(\mathsf{pp}, s; r) \to Z$**.** *Given a value* $s \in \mathcal{S} \subset \mathbb{Z}_N$*, use randomness* $r \in \mathbb{Z}_N$ *to compute and output*

$$Z := (g^r \mod N,\ h^r \cdot y^s \mod N) \in \mathbb{Z}_N^* \times \mathbb{Z}_N^*$$

$\mathsf{HTLP.Open}(\mathsf{pp}, Z, r) \to s$**.** *Parse* $Z := (u, v)$ *and compute* $w := u^{2^T} \mod N = h^r$ *via repeated squaring. Compute* $S := v/w$ *and brute force the discrete logarithm of* S *w.r.t.* y *to obtain* s*.*

$\mathsf{HTLP.Eval}(\mathsf{pp}, f, Z_1, Z_2) \to Z$**.** *To evaluate a linear function* $f(x_1, x_2) = b + a_1 x_1 + a_2 x_2$ *homomorphically on puzzles* $Z_1 := (u_1, v_1)$ *and* $Z_2 := (u_2, v_2)$*, return*

$$Z = (u_1^{a_1} \cdot u_2^{a_2} \mod N, v_1^{a_1} \cdot v_2^{a_2} \cdot y^b \mod N).$$

Figure 7: Efficient linear HTLP for small solution space.

while maintaining their secrecy. This prevents users from "poisoning" the aggregate HTLP maintained by the on-chain coordinator. For efficiency, we make use of custom NIZKs (see **??**).

Since submissions will be instantiated as HTLPs in our application and all known HTLP constructions use groups of unknown order, our proofs of well-formedness must also operate over these groups. Previous ballot correctness proofs [Gro05] and sigma protocols [Sch90, CP93] generally operate in groups of prime order and cannot directly be applied in groups of unknown order [BCM05]. To circumvent these impossibility results, we follow the blueprint of [BBF19] and instantiate our protocols in generic groups of unknown order [DK02] with a common reference string. We detail our protocols in **??**.

Our protocols are both practically efficient, private until the end of the voting/bidding phase, and provably secure, overcoming the following challenges:

4.3 System Model

Our system design is illustrated in Figure 8. We envision three types of participants:

Users. We simply refer to voters or bidders as *users*. Users submit bids or ballots, which we generically call *submissions*. We assume some external process to establish the set of authorized users (which may be open to all).

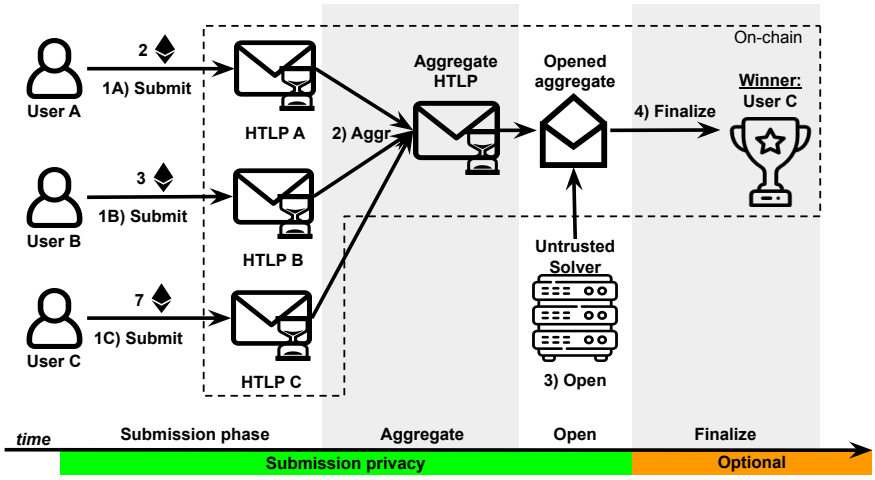

Figure 8: Our model of an HTLP-based auction/voting scheme. *(1) Submission phase:* users generate their bids/ballots as HTLPs and post them to a public bulletin board, e.g., a blockchain. *(2) Aggregation:* an on-chain contract homomorphically combines submissions into a single aggregate puzzle. *(3) Opening:* after the submissions have been aggregated, an off-chain entity solves the aggregate HTLP using T sequential steps and submits the solution to the contract. *(4) Finalize:* The smart contract may do some final computation over the solution to compute the result and announces the winner. Submission privacy is ensured only until the start of the Open phase. In Section 4.10.1, we show how voters can optionally achieve everlasting ballot privacy.

Once users place their submissions, no further action is required of them.

On-chain coordinator. We refer to the tallier/auctioneer as the *coordinator*, typically implemented as a smart contract that collects submissions. The coordinator transparently calculates the winner(s). In the case of an auction, they might also transfer (digital) assets to the winner(s). In an election, they might grant special privileges to the winner.

Off-chain solver. Since our protocols apply HTLPs, we assume an untrusted *solver* who unlocks the final HTLP(s) off-chain and submits the solution(s) to the coordinator with proofs of correct opening attached. This could, in principle, be any party, although, in practice, it will likely be one of the parties participating in (or administering) the vote/auction, or a paid marketplace [Aba23, TGB+21].

An adversary may attempt to read ballots/bids before the submission phase is complete. This is prevented by the security properties of (H)TLPs (**??**) assuming the delay T is longer than the submission phase.

	Submission domain	Hamming wt	Norm
Voting schemes			
First-past-the-post	$[0,1]^m$	1	1
Approval	$[0,1]^m$	$\leq m$	$\leq m$
Range	$[0,w]^m$	$\leq m$	$\leq wm$
Cumulative	$[0,w]^m$	$\leq m$	$\leq w$
Ranked-choice (Borda)	$\pi([0,m-1])$	$m-1$	$m(m-1)/2$
Quadratic (**??**)	$[0,\sqrt{w}]^m$	$\leq m$	$\|\mathbf{b}\|_2^2 = \langle \mathbf{b}, \mathbf{b} \rangle = w$
Single-item sealed-bid auction	$[0,w]$	1	$\leq w$
Bayesian truth serum (**??**)	$[0,1]^m \times \mathbb{N}^m$	$1, 1$	$1, \leq m$

Table 4: Requirements for the domain, Hamming weight, and norm of a vector $\mathbf{b}$ in order for it to be a valid submission in various voting/auction schemes. $\pi(S)$ denotes the set of permutations of S. The norm is an ℓ_1 norm unless otherwise specified. m is the number of candidates and w is the maximum weight which can be assigned to any one candidate.

4.4 Voting and auction schemes

We recall the specifics of FPTP, approval, range, and cumulative voting, along with single-item sealed bid auctions. The cryptographically relevant details of these schemes (i.e., the valid ballots' structure: their domain, Hamming weight, and norm) are summarized in Table 4. In **??**, we create private voting protocols for these schemes of interest.

Majority, approval, range, and cumulative voting. In the classic first-past-the-post (FPTP) voting scheme, voters can cast a vote of 1 (support) for one candidate and 0 for all others. A slight generalization of FPTP is approval voting, where users can assign a 1 vote to multiple candidates, i.e., the cast ballot s can be seen as $s \in \{0,1\}^m$, where m is the number of causes. A further generalization is range voting, where users can give each candidate up to some weight w (thus, approval is the special case where $w = 1$). A related scheme is cumulative voting, where users can distribute a total of w votes (points) among the candidates (now FPTP is a special case where $w = 1$). In each case, each candidate's points are tallied and the candidate with the highest number is declared the winner.

Ranked-choice voting. In a ranked-choice voting scheme, voters can signal more fine-grained preferences among m candidates by listing them in order of preference. There are multiple approaches to determining the winner, including single transferrable vote (STV) and instant runoff voting (IRV). In this work, we focus on the simpler Borda count version [Eme13], where each voter can cast $m-1$ points to their first-choice candidate, $m-2$ points to their second-choice

candidate, etc., and the candidate with the most points is the winner. Our protocols can be adapted to similar counting functions, such as the Dowdall system [FG14], via minor modifications.

Quadratic voting. In quadratic voting [LW18], each user's ballot is a vector $\mathbf{b} = (b_1, \ldots, b_m)$ such that $\langle \mathbf{b}, \mathbf{b} \rangle = \|\mathbf{b}\|_2^2 \leq w$. Once again, the winner is determined by summing all the ballots and determining the candidate with the most points. Thus, this is also an additive voting scheme. However, proving ballot well-formedness efficiently in this particular case benefits greatly from the novel application of the residue numeral system (RNS) to private voting (see Section 2.4).

Single-item sealed-bid auction. In a sealed-bid auction for a single item (e.g., an NFT or domain name), users submit secret bids to the auction contract. The domain of the bids might be constrained, e.g., $b \in \{0,1\}^k$ (in our implementations $k \approx 8 - 16$; see Section 4.9). Therefore, bidders must prove that their bid is well-formed, i.e., falls into that domain. Once all secret bids are revealed, the contract selects the highest bidder and awards them the auctioned item. The price the winner must pay depends on the auction scheme: e.g., highest bid in a first price auction, second-highest in a Vickrey auction.

4.5 Formalizing Time-Locked Voting and Auction Protocols

We introduce a generic syntax for a time-locked voting/auction protocol. Any such protocol is defined for a base *scoring function* $\Sigma : \mathcal{X}^n \to \mathcal{Y}$ (e.g., second-price auction, range voting), which takes as input n submissions (bids/ballots) $s_1, \ldots, s_n$ in the submission domain $\mathcal{X}$ and computes the election/auction result $\Sigma(s_1, \ldots, s_n) \in \mathcal{Y}$. It is useful to break down the scoring function into the "tally" or aggregation function $t : \mathcal{X}^n \to \mathcal{X}'$ and the finalization function $f : \mathcal{X}' \to \mathcal{Y}$, i.e., $\Sigma = f \circ t$. For example, in first-past-the-post voting, the tally function t is addition, and the finalization function f is arg max over the final tally/bids.

Definition 11 (Time-locked voting/auction protocol)**.** *A time-locked voting/auction protocol* $\Pi_\Sigma = (\mathsf{Setup}, \mathsf{Seal}, \mathsf{Aggr}, \mathsf{Open}, \mathsf{Finalize})$ *is defined with respect to a base voting/auction protocol* $\Sigma = f \circ t$*, where* $t : \mathcal{X}^n \to \mathcal{X}'$ *and* $f : \mathcal{X}' \to \mathcal{Y}$*.*

$\mathsf{Setup}(1^\lambda, T) \xrightarrow{\$} (\mathsf{pp}, \mathcal{Z})$**.** *Given a security parameter* λ *and a time parameter* T*, output public parameters* pp *and an initial time-locked value* $\mathcal{Z}$*.*

$\mathsf{Seal}(\mathsf{pp}, i, s) \xrightarrow{\$} (\mathcal{Z}_i, \pi_i)$**.** *User* $i \in [n]$ *seals its submission* $s \in \mathcal{X}$ *into a time-locked submission* $\mathcal{Z}_i$*. It also outputs a proof of well-formedness* π_i*.*

$\mathsf{Aggr}(\mathsf{pp}, \mathcal{Z}, i, \mathcal{Z}_i, \pi_i) \to \mathcal{Z}'$**.** *Given a time-locked running computation* $\mathcal{Z}$*, time-locked submission* $\mathcal{Z}_i$ *of user* i*, and proof* π_i*, the transparent contract checks the proof and potentially updates* $\mathcal{Z}$ *to* $\mathcal{Z}'$*.*

$\mathsf{Open}(\mathsf{pp}, \mathcal{Z}) \to (\mathbf{s}, \pi_{\mathsf{open}})$. *Open* $\mathcal{Z}$ *to* $\mathbf{s} = t(s_1, \ldots, s_n)$*, requiring* T *sequential steps, and compute a proof* π_{open} *to prove correctness of* $\mathbf{s}$*.*

$\mathsf{Finalize}(\mathsf{pp}, \mathcal{Z}, \mathbf{s}, \pi_{\mathsf{open}}) \to \{y, \bot\}$. *Given proposed opening* $\mathbf{s}$ *of* $\mathcal{Z}$ *and proof* π_{open}*, the coordinator may reject* $\mathbf{s}$ *or compute the final result* $y = f(\mathbf{s}) \in \mathcal{Y}$*.*

We note that the $\mathsf{Setup}(\cdot)$ algorithm in our protocols may use private randomness. In particular, our constructions use cryptographic groups (RSA and Paillier groups) that cannot be efficiently instantiated without a trusted setup (an untrusted setup would require gigantic moduli [San99]). This trust can be minimized by generating the group via a distributed trusted setup, e.g., [BF01, CHI+21, DM10]. Alternatively, the HTLPs in our protocols could be instantiated in class groups [TCLM21], which do not require a trusted setup; however, HTLPs in class groups are less efficient and verifying them on-chain would be prohibitively costly (see **??**).

A time-locked voting/auction protocol Π_Σ must satisfy the following three security properties:

Correctness. Π_Σ is *correct* if, assuming setup, submission of n puzzles, aggregation of all n submissions, and opening are all performed honestly, $\mathsf{Finalize}$ outputs a winner consistent with the base protocol Σ.

Definition 12 (Correctness)**.** *We say a voting/auction protocol* Π_Σ *with* $\Sigma : \mathcal{X}^n \to \mathcal{Y}$ *is* correct *if for all* $T, \lambda \in \mathbb{N}$ *and submissions* $s_1, \ldots, s_n \in \mathcal{X}$*,*

$$\Pr\left[\begin{array}{c|c} \mathsf{Finalize}(\mathsf{pp}, \mathcal{Z}_{\mathsf{final}}, \mathcal{S}, \pi_{\mathsf{open}}) & (\mathsf{pp}, \mathcal{Z}) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, T) \ \wedge \\ = \Sigma(s_1, \ldots, s_n) & (\mathcal{Z}_i, \pi_i) \leftarrow\!\!\$\ \mathsf{Seal}(\mathsf{pp}, i, s_i)\ \forall i \in [n] \ \wedge \\ & \mathcal{Z}_{\mathsf{final}} \leftarrow \mathsf{Aggr}(\mathsf{pp}, \mathcal{Z}, \{i, \mathcal{Z}_i, \pi_i\}_{i\in[n]}) \ \wedge \\ & (\mathcal{S}, \pi_{\mathsf{open}}) \leftarrow \mathsf{Open}(\mathsf{pp}, \mathcal{Z}_{\mathsf{final}}) \end{array}\right] = 1$$

where the aggregation step is performed over all n *submissions in any order.*

Submission privacy. The scheme satisfies *submission privacy* if the adversary cannot distinguish between two submissions, i.e., bids or ballots. Note that this property is only ensured up to time T.

Definition 13 (Submission privacy)**.** *We say that a voting/auction protocol* Π_Σ *with* $\Sigma : \mathcal{X}^n \to \mathcal{Y}$ *is* submission private *if for all* $T, \lambda \in \mathbb{N}, i \in [n]$ *and all PPT adversaries* $\mathcal{A}$ *running in at most* T *sequential steps, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\Pr\left[b = b' \left| \begin{array}{c} (\mathsf{pp}, \mathcal{Z}) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, T) \ \wedge \\ b \leftarrow\!\!\$\ \{0,1\} \ \wedge \\ (\mathcal{Z}_i, \pi_i) \leftarrow\!\!\$\ \mathsf{Seal}(\mathsf{pp}, i, b) \ \wedge \\ b' \leftarrow \mathcal{A}(\mathsf{pp}, \mathcal{Z}, i, \mathcal{Z}_i, \pi_i) \end{array}\right.\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Non-malleability. Notice that submission privacy alone does not suffice for security: even without knowing the contents of other puzzles, an adversary could submit a value that depends on other participants' (sealed) submissions. For example, in an auction, one could be guaranteed to win by homomorphically computing an HTLP containing the sum of all the other participants' bids plus a small value ε. Therefore, we also require *non-malleability*, which requires that no participant can take another's submission and replay it or "maul" it into a valid submission under its own name.

Definition 14 (Non-malleability)**.** *We say that a voting/auction protocol* Π_Σ *with* $\Sigma : \mathcal{X}^n \to \mathcal{Y}$ *is* non-malleable *if for all* $T, \lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$ *running in at most* T *sequential steps, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that the following probability is bounded by* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{c|c} \mathsf{Aggr}(\mathsf{pp}, \mathcal{Z}, i, \mathcal{Z}_i, \pi_i) \neq \mathcal{Z} \ \wedge & (\mathsf{pp}, \mathcal{Z}) \leftarrow\!\$\ \mathsf{Setup}(1^\lambda, T) \ \wedge \\ (i, \cdot, \mathcal{Z}_i, \pi_i) \notin \mathcal{Q} & (i, \mathcal{Z}_i, \pi_i) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Seal}}(\mathsf{pp}, \cdot, \cdot)}(\mathsf{pp}, \mathcal{Z}) \end{array}\right]$$

where $\mathcal{O}_{\mathsf{Seal}}(\mathsf{pp}, \cdot, \cdot)$ *is an oracle which takes as input any* $j \in [n]$ *and* $s_j \in \mathcal{X}$ *and outputs* $(\mathcal{Z}_j, \pi_j) \leftarrow\!\$\ \mathsf{Seal}(\mathsf{pp}, j, s_j)$*, and* $\mathcal{Q}$ *is the set of queries and responses* $(j, s_j, \mathcal{Z}_j, \pi_j)$ *to the oracle.*

A note on anonymity. We consider user anonymity an orthogonal problem. In the applications we have in mind, users can increase their anonymity by using zero-knowledge mixers [PSS19] or other privacy-enhancing overlays, e.g., zero-knowledge sets [Eth19]. Additionally, users can decouple their identities from their ballots by applying a verifiable shuffle [Nef01], although the on-chain verification of a shuffle proof might be prohibitively costly for larger elections. In Section 4.10.1 we describe how our protocols can be extended to achieve bid privacy even after the election ends, thus disclosing nothing besides a user's (non-)participation.

4.6 The Cicada Framework

We present Cicada, our framework for non-interactive private auctions/elections, in Figure 9. Cicada can be applied to voting and auction schemes where the scoring function $\Sigma = f \circ t$ has a linear tally function t. The framework uses a linear HTLP (Section 2.2), vector packing scheme (Section 2.4), and matching NIZK (Section 4.7) to ensure correctness of submissions by proving both the well-formedness of the puzzle and the solution's membership in $\mathcal{X}$.

At a high level, Cicada enables auction or voting schemes with the following five steps. *First*, system participants agree on a delay parameter T and packing parameter ℓ. The Setup algorithm outputs HTLP public parameters pp (note this might require private randomness) and initializes a set of the tally HTLPs $\mathcal{Z}$ containing zeros. *Second*, a user i uses the Seal algorithm to encode their submission (a bid or ballot) $\mathbf{v}_i$ into (a set of) HTLP(s) $\mathcal{Z}_i$. Seal also outputs a NIZK π_i proving that the submission $\mathcal{Z}_i$ is well-formed, i.e., is in the domain $\mathcal{X}$ of the scoring function Σ. Users will send these submissions and corresponding proofs

to the on-chain coordinator. *Third*, the coordinator (Cicada smart contract) runs Aggr to verify the user-submitted proof π_i, and if it is valid, aggregate the user submission $\mathcal{Z}_i$ into the tally HTLPs $\mathcal{Z}$, resulting in updated tally HTLP(s) $\mathcal{Z}'$. *Fourth*, after the voting/bidding period has ended, any party can open the tally HTLPs $\mathcal{Z}$ off-chain by running Open, which outputs the opening(s) $\mathcal{S}$ of the tally HTLP(s) $\mathcal{Z}$ along with a proof of correct opening π_{open} (using a proof of solution PoS, see Section 4.7). The off-chain solver will send $\mathcal{S}, \pi_{\mathsf{open}}$ to the contract. *Fifth*, the on-chain contract runs $\mathsf{Finalize}$ to verify the correctness of $\mathcal{S}$ by checking π_{open}. If the check passes, it computes the auction/election winner(s) as $y = f(\mathbf{v})$.

Intuitively, submission privacy follows from the security of the HTLP and zero-knowledge of the NIZK used: the submission can't be opened before time T and none of the proofs leak any information about it. Non-malleability is enforced by requiring the NIZK to be a proof of knowledge and including the user's identity i in the instance to prove, e.g., including it in the hash input of the Fiat-Shamir transform. This prevents a malicious actor from replaying a different user's ballot correctness proof.

As we will see next, this captures many common schemes such as cumulative voting and sealed-bid auctions. Cicada introduces a crucial design choice via the packing parameter $\ell \in [m]$, which defines a storage-computation trade-off that we detail in Section 4.9.

Additive voting. Many common voting schemes are "additive", meaning each ballot (a length-m vector) is simply added to the tally, and a finalization function f is applied to the tally after the voting phase has ended to determine the winner. Additive voting schemes include first-past-the-post (FPTP), approval, range, and cumulative voting. Simple ranked-choice voting schemes, e.g., Borda count [Eme13], are also additive, differing only in what qualifies as a "proper" ballot (restrictions on vector entry domain, vector norm, etc.; see Table 4). Thus, we can use Cicada to instantiate private voting protocols for all these schemes.

Theorem 2. *Given a linear scoring function Σ, a secure NIZKPoK* NIZK *and proof of solution* PoS*, a secure* HTLP*, and a packing scheme* (PSetup, $\mathsf{Pack}, \mathsf{Unpack}$)*, the Cicada protocol Π_Σ (Figure 9) is a secure time-locked voting/auction protocol.*

Proof. For simplicity, we give a proof for the simple case of $\mathcal{X} = [0, 1]$, i.e., submissions consist of a single bit, but our argument generalizes to larger domains $\mathcal{X}$. Let $n \in \mathbb{N}$ be the number of users.

The correctness of the Cicada framework (cf. Definition 12) follows by construction and from the correctness of the underlying building blocks (i.e., soundness in the case of the proof systems).

Next, we prove submission privacy. Let $\mathsf{ExpSPriv}^{\mathcal{A}}_{\Pi_\Sigma}(\lambda, T, i)$ be the original submission privacy game for the Cicada scheme Π_Σ with T-bounded adversary

The Cicada Framework

Let $\Sigma : \mathcal{X}^n \to \mathcal{Y}$ be the scoring function of a voting/auction scheme where $\Sigma = f \circ t$ for a linear function t and $\mathcal{X} = [0, w]^m$. Let HTLP a linear HTLP, $T \in \mathbb{N}$ a time parameter representing the election/auction length, and $(\mathsf{PSetup}, \mathsf{Pack}, \mathsf{Unpack})$ a packing scheme. Let NIZK be a NIZKPoK for submission correctness (language depends on Σ, HTLP; see Section 4.7) and PoS be a proof of correct HTLP solution (see Section 4.7).

$\mathsf{Setup}(1^\lambda, T, \ell) \xrightarrow{\$} (\mathsf{pp}, \mathcal{Z})$**.** Set up the public parameters $\mathsf{pp}_{\mathsf{NIZK}} \leftarrow\!\!\$\ \mathsf{NIZK.Setup}(1^\lambda)$, $\mathsf{pp}_{\mathsf{tlp}} \leftarrow\!\!\$\ \mathsf{HTLP.Setup}(1^\lambda, T)$, and $\mathsf{pp}_{\mathsf{pack}} \leftarrow \mathsf{PSetup}(\ell, w)$. Let $\mathcal{Z} = \{Z_j\}_{j\in[m/\ell]}$ where $Z_j \leftarrow\!\!\$\ \mathsf{HTLP.Gen}(0)$. Output $\mathsf{pp} := (\mathsf{pp}_{\mathsf{tlp}}, \mathsf{pp}_{\mathsf{pack}}, \mathsf{pp}_{\mathsf{NIZK}})$ and $\mathcal{Z}$.

$\mathsf{Seal}(\mathsf{pp}, i, \mathbf{v}_i) \xrightarrow{\$} (\mathcal{Z}_i, \pi_i)$**.** Parse $\mathbf{v}_i := \mathbf{v}_{i,1}||\ldots||\mathbf{v}_{i,m/\ell}$. Compute $Z_{i,j} \leftarrow \mathsf{HTLP.Gen}(\mathsf{Pack}(\mathbf{v}_{i,j}))\ \forall j \in [m/\ell]$ and $\pi_i \leftarrow \mathsf{NIZK.Prove}((i, \mathcal{Z}_i), \mathbf{v}_i)$. Output $(\mathcal{Z}_i := \{Z_{i,j}\}_{j\in[m/\ell]}, \pi_i)$

$\mathsf{Aggr}(\mathsf{pp}, \mathcal{Z}, i, \mathcal{Z}_i, \pi_i) \to \mathcal{Z}'$**.** If $\mathsf{NIZK.Vrfy}((i, \mathcal{Z}_i), \pi_i) = 1$, update $\mathcal{Z}$ to $\mathcal{Z} \boxplus \mathcal{Z}_i$.

$\mathsf{Open}(\mathsf{pp}, \mathcal{Z}) \to (\mathcal{S}, \pi_{\mathsf{open}})$**.** Parse $\mathcal{Z} := \{Z_j\}_{j\in[m/\ell]}$ and solve for the encoded tally $\mathcal{S} = \{s_j\}_{j\in[m/\ell]}$ where $s_j \leftarrow \mathsf{HTLP.Solve}(Z_j)$. Prove the correctness of the solution(s) as $\pi_{\mathsf{open}} \leftarrow \mathsf{PoS.Prove}(\mathcal{S}, \mathcal{Z}, 2^T)$ and output $(\mathcal{S}, \pi_{\mathsf{open}})$.

$\mathsf{Finalize}(\mathsf{pp}, \mathcal{Z}, \mathcal{S}, \pi_{\mathsf{open}}) \to \{y, \perp\}$**.** If $\mathsf{PoS.Verify}(\mathcal{S}, \mathcal{Z}, 2^T, \pi_{\mathsf{open}}) \neq 1$, return $\perp$. Otherwise, parse $S := \{s_j\}_{j\in[m/\ell]}$ and let $\mathbf{v} := \mathbf{v}_1||\ldots||\mathbf{v}_{m/\ell}$, where $\mathbf{v}_j \leftarrow \mathsf{Unpack}(s_j)\ \forall j \in [m/\ell]$. Output y such that $y = \Sigma(\mathbf{v})$.

Figure 9: The Cicada framework for non-interactive private auctions and elections.

$\mathcal{A}$, cf. Definition 13. We define a series of hybrids to show that

$$\Pr[\mathsf{ExpSPriv}^{\mathcal{A}}_{\Pi_\Sigma}(\lambda, T, i) = 1] \leq \mathsf{negl}(\lambda)$$

for all $\lambda, T \in \mathbb{N}$ and $i \in [n]$.

$\underline{\mathcal{H}_0}$: This is the original game $\mathsf{ExpSPriv}^{\mathcal{A}}_{\Pi_\Sigma}(\lambda, T, i)$, where $Z_i \leftarrow \mathsf{HTLP.Gen}(b)$ and $\pi_i \leftarrow \mathsf{NIZK.Prove}(i, Z_i, b)$.

$\underline{\mathcal{H}_1}$: Replace π with $\tilde{\pi} \leftarrow \mathsf{NIZK}.\mathcal{S}(i, Z_i)$. $\mathcal{H}_1$ is indistinguishable from $\mathcal{H}_0$ by the zero-knowledge property of NIZK.

$\underline{\mathcal{H}_2}$: Replace Z_i with $Y_i \leftarrow \mathsf{HTLP.Gen}(1-b)$ and $\tilde{\pi}$ with $\tilde{\sigma} \leftarrow \mathsf{NIZK}.\mathcal{S}(i, Y_i)$. $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable because the distributions $\{Z_i, \mathcal{S}(i, Z_i)\}$ and $\{Y_i, \mathcal{S}(i, Y_i)\}$ are indistinguishable since $\{Z_i\}, \{Y_i\}$ are indistinguishable by the security of HTLP.

$\mathcal{H}_3$: Replace $\tilde{\sigma}$ with $\sigma \leftarrow \mathsf{NIZK.Prove}(i, Y_i, 1-b)$. $\mathcal{H}_3$ is indistinguishable from $\mathcal{H}_2$ by the zero-knowledge property of NIZK.

This series of hybrids implies $\Pr[b'=b] \approx_\lambda \Pr[b'=1-b]$, where b' is the output of $\mathcal{A}$ in $\mathcal{H}_0$ or $\mathcal{H}_3$, respectively. Therefore $\Pr[\mathsf{ExpSPriv}^{\mathcal{A}}_{\Pi_\Sigma}(\lambda, T, i) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

Finally, we show that if NIZK is a PoK and HTLP is secure, then Cicada is non-malleable, cf. Definition 14. Suppose towards a contradiction that Cicada is malleable. We will use this and the fact that NIZK is a PoK to construct an adversary $\mathcal{B}$ which has non-negligible advantage in the HTLP security game. Again, we work in the simple case $\mathcal{X} = [0,1]$, i.e., $m, \ell, w = 1$, but the argument generalizes to other parameter settings.

Since by our assumption Cicada is malleable, there exists $\mathcal{A}$ which outputs $(i, \cdot, \mathcal{Z}_i, \pi_i) \notin \mathcal{Q}$ such that $\mathsf{NIZK.Vrfy}((i, \mathcal{Z}_i), \pi) = 1$ with non-negligible probability. Given a puzzle Z_b containing some unknown bit b, $\mathcal{B}$ works as follows. First, it computes $(\mathsf{pp}, Z) \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, T, 1)$ and sends them to the non-malleability adversary $\mathcal{A}$. $\mathcal{B}$ responds to $\mathcal{A}$'s oracle queries (j, b_j) with honestly computed (Z_j, π_j), keeping track of queries and responses in the set $\mathcal{Q}$. When $\mathcal{A}$ outputs (i, Z_i, π_i), $\mathcal{B}$ looks for $(i, b_i, Z_i, \pi_i) \in \mathcal{Q}$ and outputs b_i. Since $\mathcal{A}$ has non-negligible advantage, it follows that $\mathsf{NIZK.Vrfy}((i, Z_i), \pi_i) = 1$. This implies that either $\Pr[b_i = b] = \frac{1}{2} + \mathsf{negl}(\lambda)$ or NIZK is not knowledge sound. Both possibilities contradict our assumptions, namely that the HTLP is secure and the NIZK is knowledge sound. Thus, Cicada must be non-malleable. □

4.7 Ballot/bid correctness proofs

For our proofs, we assume HTLPs are of the form $(u, v) = (g^r, h^r y^s) \in \mathbb{G}_1 \times \mathbb{G}_2$, where $\mathbb{G}_1, \mathbb{G}_2$ are groups of unknown order. This captures all known constructions of HTLPs: in the case of the Paillier HTLP (Construction 1), $\mathbb{G}_1 = \mathbb{J}_N$, $\mathbb{G}_2 = \mathbb{Z}^*_{N^2}$, $h = (g^{2^T})^N$, and $y = 1 + N$. For the exponential ElGamal HTLP (Construction 10), $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{Z}^*_N$, $h = g^{2^T}$, and $y \in \mathbb{G}_1$. And for the class group HTLP [TCLM21], $\mathbb{G}_1, \mathbb{G}_2$ are cyclic subgroups of the respective class groups $Cl(\Delta_K), Cl(q^2\Delta_K)$, respectively, $h = \psi_q(G^{2^T})$ where G is a generator of $\mathbb{G}_1$ and $\psi_q : Cl(\Delta_K) \to Cl(q^2\Delta_K)$ is an injective map, and $y \in \mathbb{G}_2$ is the generator of a subgroup in which the discrete logarithm problem is easy (see [TCLM21] for details).

Proof of solution. During the finalization phase of our protocol, any party can solve the final HTLP off-chain and submit a solution to the contract. To enforce the correctness of this solution we require the solver to include a proof of the following relation:

$$\mathcal{R}_{\mathsf{PoS}} = \{((y, u, v, w \in \mathbb{G}, s \in \mathbb{Z}); \perp) : w = u^{2^T} \wedge\ v = wy^s \in \mathbb{G}\} \tag{5}$$

We call such a proof system $\mathsf{PoS} = (\mathsf{Prove}, \mathsf{Verify})$. It can be realized as the conjunction of two proofs of exponentiation (PoE) [Pie19, Wes19] for $w = u^{2^T}$

and $y^s = v/w$. A PoE is a proof for the following relation:

$$\mathcal{R}_{\mathsf{PoE}} = \{((u, w \in \mathbb{G}, x \in \mathbb{Z}); \perp) : w = u^x \in \mathbb{G}\}$$

Note that there is no witness in the $\mathcal{R}_{\mathsf{PoE}}$ relation, i.e., the verifier knows the exponent x. The primary goal of the PoE proof system for the verifier is to outsource a possibly large exponentiation in a group $\mathbb{G}$ of unknown order.

Wesolowski's proof of exponentiation protocol (PoE)

Public parameters: $\mathbb{G} \leftarrow\!\!\$\ GGen(\lambda)$.
Public inputs: $u, w \in \mathbb{G}, x \in \mathbb{Z}$.
Claim: $u^x = w$.

1. $\mathcal{V}$ sends $l \leftarrow\!\!\$\ \mathsf{Primes}(\lambda)$ to $\mathcal{P}$.
2. $\mathcal{P}$ computes $q = \lfloor \frac{x}{l} \rfloor \in \mathbb{Z} \wedge r \in [l]$, where $x = ql + r$. $\mathcal{P}$ sends $Q = u^q \in \mathbb{G}$ to $\mathcal{V}$.
3. $\mathcal{V}$ computes $r = x \bmod l$.

$\mathcal{V}$ accepts iff $w = Q^l u^r$.

Observe that the verifier sends a prime number as a challenge. When we make this protocol non-interactive via the Fiat-Shamir transform, we use a standard $\mathsf{HashToPrime}(\cdot)$ function to generate the correct challenge for the prover. In our implementation, we use the Baillie-PSW primality test [PSW80] to show that a randomly hashed challenge is indeed prime.

Proofs of well-formedness. To prove that HTLP ballots are well-formed during the submission phase, we will use several different proofs of knowledge about TLP solutions. Most of our protocols make use of the fact that for such HTLPs, v has the same structure as a Pedersen commitment [Ped92].

Since we are operating in groups of unknown order, to circumvent the impossibility result of [BCK10] and achieve negligible soundness error for Schnorr-style sigma protocols, we assume access to some public element(s) of $\mathbb{G}_1, \mathbb{G}_2$ whose representations are unknown. We prove security assuming $\mathbb{G}_1, \mathbb{G}_2$ are generic groups output by some randomized algorithm $GGen(\lambda)$. For more on instantiating Schnorr-style protocols in groups of unknown order while maintaining negligible soundness error, see [BBF19].

Well-formedness and knowledge of solution. To prove knowledge of a puzzle solution in zero-knowledge, our starting point is the folklore Schnorr-style protocol for knowledge of a Pedersen-committed value. Our protocol zk-PoKS is shown below.

zkPoK of TLP solution (zk-PoKS)

Public parameters: $\mathbb{G}_1, \mathbb{G}_2 \leftarrow\$\ GGen(\lambda)$, $b > 2^{2\lambda}|\mathbb{G}_i|\ \forall i \in \{1,2\}$, and $g \in \mathbb{G}_1, h, y \in \mathbb{G}_2$.
Public input: HTLP $Z = (u, v)$.
Private input: $s, r \in \mathbb{Z}$ such that $Z = (g^r, h^r y^s)$.

1. $\mathcal{P}$ samples $\alpha, \beta \leftarrow\$\ [-b, b]$ and sends $A := h^\alpha y^\beta, B := g^\alpha$ to $\mathcal{V}$.
2. $\mathcal{V}$ sends a challenge $e \leftarrow\$\ [2^\lambda]$.
3. $\mathcal{P}$ computes $w = re + \alpha$ and $x = se + \beta$, which it sends to $\mathcal{V}$.

$\mathcal{V}$ accepts iff the following hold:

$$v^e A = h^w y^x$$
$$u^e B = g^w$$

Equality of solutions. Again, our starting point is the folklore protocol of equality of Pedersen-committed values: given two HTLPs with second terms v_1, v_2, if the solutions are equal the quotient is $v_1/v_2 = h^{r_1 - r_2}$. To prove the equality of the solutions, it therefore suffices to show knowledge of the discrete logarithm of v_1/v_2 with respect to h using Schnorr's classic sigma protocol [Sch90] with the previously described adjustments. Because of its simplicity we do not explicitly write out the protocol, which we will refer to as zk-PoSEq.

Binary solution. In an FPTP vote for $m = 2$ candidates, users only need to prove that their ballot $(g^r, h^r y^s)$ encodes 0 or 1. More formally, users prove the statement $(u = g^r \wedge v = h^r) \vee (u = g^r \wedge vy^{-1} = h^r)$. This can be proved using the OR-composition [CDS94] of two discrete logarithm equality proofs [CP93] with respect to bases g and h and discrete logarithm r. A similar proof strategy could be applied if the user has multiple binary choices, e.g., approval and range voting. The OR-composition of multiple discrete logarithm equality proofs yields a secure ballot correctness proof for those voting schemes.

Positive solution. We use Groth's trick [Gro05], based on the classical Legendre three-square theorem from number theory, to show that a puzzle solution s is positive by showing that $4s + 1$ can be written as the sum of three squares. Our protocol deals only with the second component of the TLP, making use of the proof of solution equality (zk-PoSEq) described above and a proof that a TLP solution is the square of another (zk-PoKSqS, described next).

Proof of positive solution (zk-PoPS)

Public parameters: $\mathbb{G}_2 \leftarrow\$\ GGen(\lambda)$, a secure HTLP, and $h, y \in \mathbb{G}_2$.

Public input: $v \in \mathbb{G}_2$ such that $(\cdot, v) \in \mathsf{Im}(\mathsf{HTLP.Gen})$.
Private input: $s, r \in \mathbb{Z}$ such that $v = h^r y^s$ and $s > 0$.

1. Find three integers $s_1, s_2, s_3 \in \mathbb{Z}$ such that $4s + 1 = s_1^2 + s_2^2 + s_3^2$ and, for each $j = 1, 2, 3$, compute two HTLPs:

$$Z_j \leftarrow \mathsf{HTLP.Gen}(s_j)$$
$$Z'_j \leftarrow \mathsf{HTLP.Gen}(s_j^2)$$

2. Use zk-PoKSqS to compute a proof σ_j of square solution for each pair (Z_j, Z'_j) for $j = 1, 2, 3$.

3. Use zk-PoSEq to compute a proof σ_{eq} of solution equality for $4 \cdot Z \boxplus 1$ and $Z'_1 \boxplus Z'_2 \boxplus Z'_3$.

The full proof consists of $(\sigma_1, \sigma_2, \sigma_3, \sigma_{\mathsf{eq}})$, all computed with the same challenge $e \in [2^\lambda]$.

Square solution. To prove that a puzzle solution is the square of another, we use a conjunction of two zk-PoKS variants which proves knowledge of the same solution with respect to different bases. In particular, we consider only the second terms $v_1 = h^{r_1} y^s$ and $v_2 = h^{r_2} y^{s^2}$. We use the fact that v_2 can be rewritten as $h^{r_2 - r_1 s} v_1^s$ and prove that its opening w.r.t. base v_1 equals the opening of v_1.

Proof of square solution (zk-PoKSqS)

Public parameters: $\mathbb{G}_2 \leftarrow\!\!\$\ GGen(\lambda)$, $b > 2^{2\lambda}|\mathbb{G}_2|$, and $h, y \in \mathbb{G}_2$.
Public input: $v_1, v_2 \in \mathbb{G}_2$.
Private input: $s, r_1, r_2 \in \mathbb{Z}$ such that $v_1 = h^{r_1} y^s$ and $v_2 = h^{r_2} y^{s^2} = h^{r_2 - r_1 s} v_1^s$.

1. $\mathcal{P}$ samples $\alpha_1, \alpha_2, \beta \leftarrow\!\!\$\ [-b, b]$ and sends $A_1 := h^{\alpha_1} y^\beta, A_2 := h^{\alpha_2} v_1^\beta$ to $\mathcal{V}$.

2. $\mathcal{V}$ sends a challenge $e \leftarrow\!\!\$\ [2^\lambda]$.

3. $\mathcal{P}$ computes $w_1 = r_1 e + \alpha_1, w_2 = (r_2 - r_1 s)e + \alpha_2$, and $x = se + \beta$, which it sends to $\mathcal{V}$.

$\mathcal{V}$ accepts iff the following hold:

$$v_1^e A_1 = h^{w_1} y^x$$
$$v_2^e A_2 = h^{w_2} v_1^x$$

Borda ballot correctness

todo: incomplete

1. $\mathcal{P}$ sends $(g^a \bmod N, (1+N)^a \bmod N^2)$ for $a \leftarrow\!\!\$\ \mathbb{Z}_N$.
2. $\mathcal{V}$ sends a challenge $e \leftarrow\!\!\$\ [1, n]$.
3. $\mathcal{P}$ sends $(g^{r\cdot e+a}, h^{r\cdot N\cdot e+a}(1+N)^{s\cdot e+a} \bmod N^2)$. $\mathcal{V}$ accepts iff the following hold:

Quadratic voting [LW18]. Each voter i submits two linear HTLPs: Z_i^{tally} containing s_i and Z_i^{norm} containing s_i^2, where s_i is an encoding of the ballot $\mathbf{b}_i$. Z_i^{tally} will be accumulated into the running tally as usual, and Z_i^{norm} will be used to enforce the norm bound. A well-formed sealed ballot is therefore of the form $Z_i = (Z_i^{\mathsf{tally}}, Z_i^{\mathsf{norm}})$ such that:

Check #1. The vector entries enclosed in Z_i^{norm} are the squares of those enclosed in Z_i^{tally}.

Check #2. Z_i^{norm} has ℓ_1 norm strictly equal to w.[19]

The first check is much simpler and more efficient when using RNS packing. Recall that with this packing, a solution s encodes the ballot $(b_1, \ldots, b_m)$ as $s \bmod p_j \equiv b_j\ \forall j \in [m]$, and that this encoding is fully SIMD homomorphic. It follows that $s^2 \bmod p_j \equiv b_j^2$ for all $j \in [m]$.[20] With the RNS packing it therefore suffices to prove a square relationship *once* for the puzzles encoding s and s^2 (e.g., using zk-PoKSqS) rather than m times for all the vector entries. This is in contrast to the PNS packing used by all previous private voting schemes in the literature, where the absence of a multiplicative homomorphism would require proving the square relationship for every vector entry *individually*.

Regardless of the vector encoding, the second check is more involved: the user needs to open a sum of vector entries (the residues) without revealing the entries (residues) themselves. One approach is for the user to commit to each vector entry in Z_i^{norm}, i.e., $a_{ij} = s_i^2 \bmod p_j$, with a Pedersen commitment, and use a variant proof of knowledge of exponent modulo p_j (PoKEMon [BBF19]) to show the commitments contain the appropriate values a_{ij}. Then, it can open the sum of the commitments. PoKEMon proofs are batchable, so the contract can verify them efficiently and check that the sum of the commitments opens to w.

Security proofs. Finally, we prove special-soundness and honest-verifier zero-knowledge (HVZK) of the above sigma protocols. Any such protocol can be

[19] We make this stricter requirement to simplify the norm check. Note that voters should be incentivized to submit such votes, since it maximizes their voting power.

[20] Assuming $s_j^2 < p_j$ for all j, which in our case will hold regardless, we set each $p_j < nw$ to avoid overflow when adding ballots and $s_j^2 \leq w < nw$.

made into a non-interactive zero-knowledge proof of knowledge (NIZKPoK) via the Fiat-Shamir transform [FS87].

Theorem 3 (zk-PoKS). *The protocol* zk-PoKS *is a special sound and HVZK proof system in the generic group model.*

Proof. For special soundness, we show that given two distinct accepting transcripts with the same first message, i.e., (A, B, e, w, x) and (A, B, e', w', x') where $e \neq e'$, we can extract the witnesses r, s. The proof follows the blueprint of the proof of Theorem 10 in [BBF19]. Since the transcripts are accepting, we have

$$\begin{aligned} h^w y^x &= v^e A & h^{w'} y^{x'} &= v^{e'} A \\ &= h^{re+\alpha} y^{se+\beta} & &= h^{re'+\alpha} y^{se'+\beta} \end{aligned}$$

Combining the two equations we get

$$\begin{aligned} & h^{r\Delta e} y^{s\Delta e} = h^{\Delta w} y^{\Delta x} \\ \iff \quad & v^{\Delta e} = h^{\Delta w} y^{\Delta x} \end{aligned} \tag{6}$$

where $\Delta e = e - e'$ and $\Delta y, \Delta x$ are defined similarly. Then with overwhelming probability, $r\Delta e = \Delta w$ and $s\Delta e = \Delta x$ (cf. Lemma 4 of [BBF19]), so $\Delta e \in \mathbb{Z}$ divides $\Delta w \in \mathbb{Z}$ and $\Delta x \in \mathbb{Z}$ and we can extract $r, s \in \mathbb{Z}$ as $r = \Delta w / \Delta e$ and $s = \Delta x / \Delta e$.

We will now show that these values are correct, i.e., $v = h^{\Delta w/\Delta e} y^{\Delta x/\Delta e}$. Assume towards a contradiction that this does not hold and $\mu = h^{\Delta w/\Delta e} y^{\Delta w/\Delta e} \neq v$. Since $\mu^{\Delta e} = v^{\Delta e}$ by Equation (6), this must mean that $(\mu/v)^{\Delta e} = 1$ and therefore $\mu/v \in \mathbb{G}_2$ is an element of order $\Delta e > 1$. Since Δe is easy to compute and μ/v is a non-identity element of $\mathbb{G}_2$, this contradicts the assumption that $\mathbb{G}_2$ is a generic group (specifically, it contradicts non-trivial order hardness [BBF19, Corollary 2]). We thus conclude that our extractor successfully recovers the witnesses r and s.

We still need to verify that the r^* we can extract from u will be consistent with the one extracted from v, i.e., $r^* = r$. Again we know

$$\begin{aligned} g^w &= u^e B & g^{w'} &= u^{e'} B \\ &= g^{r^* e + \alpha^*} & &= g^{r^* e' + \alpha^*} \end{aligned}$$

so by a similar argument $r^* = \Delta w / \Delta e$, which equals r. Thus the protocol satisfies special soundness.

To prove HVZK, we give a simulator which produces an accepting transcript $(\tilde{A}, \tilde{B}, \tilde{e}, \tilde{w}, \tilde{x})$ that is perfectly indistinguishable from an honest transcript (A, B, e, w, x). The simulator is quite simple: it samples $\tilde{e} \leftarrow\!\$\ [2^\lambda]$ identically to an honest verifier, then samples $\tilde{w}, \tilde{x} \leftarrow\!\$\ \mathbb{Z}$ and sets $\tilde{A} := h^{\tilde{w}} y^{\tilde{x}} v^{-\tilde{e}}$ and $\tilde{B} := g^{\tilde{w}} u^{-\tilde{e}}$. It follows by inspection that the transcript is an accepting one. Furthermore, notice that $\tilde{A}$ and $\tilde{B}$ are uniformly distributed in $\mathbb{G}_2$ and $\mathbb{G}_1$, respectively, just like A, B in the honest transcript. Also, both $\tilde{x}$ and x are uniform in $\mathbb{Z}$. Thus the simulated transcript is perfectly indistinguishable from an honest one. □

Theorem 4 (zk-PoKSqS). *The protocol* zk-PoKSqS *is a special sound and HVZK proof system in the generic group model.*

Proof. For special soundness, we show that given two distinct accepting transcripts with the same first message, i.e., $(A_1, A_2, e, w_1, w_2, x)$ and $(A_1, A_2, e', w_1', w_2', x')$ where $e \neq e'$, we can extract the witnesses r_1, r_2, s. Notice that v_2 is not guaranteed to encode the square of s_1, so $v_2 = h^{r_2 - r_1 s_2/s_1} v_1^{s_2/s_1}$. Let $\sigma_2 = s_2/s_1$ and $\rho_2 := r_2 - r_1 s_2/s_1 = r_2 - r_2\sigma_2$.

Using the same extractor as in the proof of Theorem 3, we can extract correct integers $r_1 = \Delta w_1/\Delta e$, $s_1 = \Delta x/\Delta e$, $\rho_2 = \Delta w_2/\Delta e$, and $\sigma_2 = \Delta x/\Delta e$. Notice $s_1 = \sigma_2$, which implies $\sigma_2 = s_1^2$. Finally we use $r_1, s_1, \rho_2 \in \mathbb{Z}$ to recover $r_2 := \rho_2 + r_1 s_1 \in \mathbb{Z}$. Thus the protocol is special sound.

To prove HVZK, we give a simulator which produces an accepting transcript $(\tilde{A}_1, \tilde{A}_2, \tilde{e}, \tilde{w}_1, \tilde{w}_2, \tilde{x})$ that is perfectly indistinguishable from an honest transcript $(A_1, A_2, e, w_1, w_2, x)$. The simulator is quite simple: it samples $\tilde{e} \leftarrow\!\!\$\ [2^\lambda]$ identically to an honest verifier, then samples $\tilde{w}_1, \tilde{w}_2, \tilde{x} \leftarrow\!\!\$\ \mathbb{Z}$ and sets $\tilde{A}_1 := h^{\tilde{w}_1} y^{\tilde{x}} v_1^{-\tilde{e}}$ and $\tilde{A}_2 := h^{\tilde{w}_2} v_1^{\tilde{x}} v_2^{-\tilde{e}}$. It follows by inspection that the transcript is an accepting one. Furthermore, notice that $\tilde{A}_1, \tilde{A}_2$ are uniformly distributed in $\mathbb{G}_2$, respectively, just like A_1, A_2 in the honest transcript. Also, both $\tilde{w}_1, \tilde{w}_2, \tilde{x}$ are uniform in $\mathbb{Z}$ just like w_1, w_2, x. Thus the simulated transcript is perfectly indistinguishable from an honest one. □

Theorem 5 (zk-PoPS). *The protocol* zk-PoPS *is sound and HVZK.*

Proof. Soundness follows directly from the (knowledge) soundness of zk-PoKSqS and zk-PoSEq as well as Legendre's three-square theorem [Gro05].

For HVZK, note that an honest zk-PoPS transcript has the form $(\{A_{1,j}, A_{2,j}\}_{j\in[3]}, R, e, \{w_{1,j}, w_{2,j}, x_j\}_{j\in[3]})$, where (R, e, x) is an honest zk-PoSEq transcript and $(A_{1,j}, A_{2,j}, e, w_{1,j}, w_{2,j}, x_j)$ for $j = 1, 2, 3$ are honest zk-PoKSqS transcripts. Given the instance v, our zk-PoPS simulator first computes some random HTLPs $(\tilde{u}_j, \tilde{v}_j), (\tilde{u}_j', \tilde{v}_j') \leftarrow\!\!\$\ \mathsf{HTLP.Gen}(0)$ for $j = 1, 2, 3$. These simulated underlying instances are indistinguishable from the honest instances an honest prover would use. This follows from the security of HTLP.

Next, our simulator samples $\tilde{e} \leftarrow\!\!\$\ [2^\lambda]$ identically to an honest verifier and uses the simulators of the proof systems, always with the same challenge $\tilde{e}$, to produce a simulated transcript:

$$(\tilde{A}_{1,j}, \tilde{A}_{2,j}, \tilde{e}, \tilde{w}_{1,j}, \tilde{w}_{2,j}, \tilde{x}_j) \leftarrow \mathcal{S}_{\mathsf{zk-PoKSqS}}(\tilde{v}_j, \tilde{v}_j'; \tilde{e})\ \forall j = 1, 2, 3$$
$$(\tilde{R}, \tilde{e}, \tilde{x}) \leftarrow \mathcal{S}_{\mathsf{zk-PoSEq}}\left(\frac{4 \cdot v \boxplus 1}{\tilde{v}_1' \boxplus \tilde{v}_2' \boxplus \tilde{v}_3'}; \tilde{e}\right)$$

By HVZK of zk-PoKSqS and zk-PoSEq, these transcripts are accepting and indistinguishable from an honestly generated transcript. □

4.8 Parameter Settings

In this section, we evaluate the practicality and optimality of various HTLP constructions based on the parameters M, n, m, w of the auction or vote. Assuming

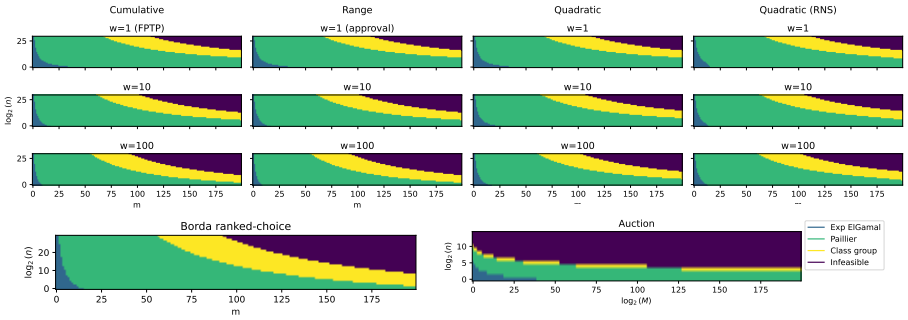

Figure 10: Most efficient HTLP construction for voting and auction using Cicada with maximal packing (using PNS except where indicated).

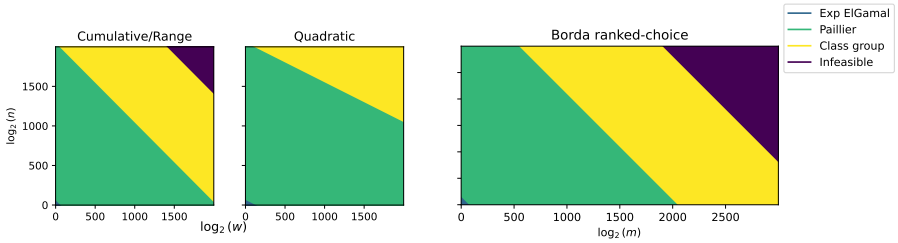

Figure 11: Most efficient HTLP construction for voting schemes using Cicada without packing. For the sealed-bid auction, the bid bit-length directly determines the HTLP construction to use (exponential ElGamal up to 80 bits, then Paillier up to 2048, and class group up to 3400).

the classic PNS packing, we require $(nw+1)^m \leq |\mathbb{G}|$ for voting and $M^n \leq |\mathbb{G}|$ for auctions, where $\mathbb{G}$ is the group in which the HTLP is instantiated. We show the optimal HTLP construction for auctions and voting for various parameter settings in Figure 10 (with packing) and Figure 11 (without packing). We use the security parameter $\lambda = 80$ (see discussion in Section 4.9), which corresponds to a 1024-bit modulus N for exponential ElGamal and Paillier HTLPs and 3400-bit discriminants for class group HTLPs. For the exponential ElGamal HTLP, we fixed the maximum ballot at 2^{80}, which corresponds to $\approx 2^{40}$ brute-forcing work using Pollard's rho algorithm [Pol78].

Exponential ElGamal HTLP (Construction 10) This is the most efficient HTLP construction: for a given security parameter, it has the smallest required cryptographic groups and most efficient group operations. However, since the puzzle solution is encoded in the exponent, solving the puzzle requires brute-forcing a discrete logarithm. This limits the use of this construction to a small set of parameter settings: assuming the largest discrete logarithm an off-chain

solver can be expected to brute-force has τ bits, we require $(nw+1)^m \leq 2^{2\tau}$.

Paillier HTLP (Construction 1) This is a slightly less efficient construction since the size of the HTLPs for a given security parameter is doubled due to working over mod N^2 instead of mod N. This increases both the required storage and the complexity of the group operation. On the other hand, due to its larger solution space, the Paillier HTLP supports much broader parameter settings for a given security parameter.

Class group HTLP Class group offer the sole HTLP construction without a trusted setup [TCLM21]. This comes at the cost of the largest groups for a given security parameter. Class groups are not widely supported by major cryptographic libraries, and their costly group operation makes blockchain deployment difficult. We are unaware of any class group implementations for Ethereum smart contracts.

Impractical parameter settings Accommodating very large settings of n, w, m, M requires larger groups, leading to group operations and storage requirements which are intolerably inefficient for certain applications.

4.9 Implementation

We implemented our transparent on-chain coordinator as an Ethereum smart contract in Solidity.[21] For efficiency, we use the exponential ElGamal HTLP with a 1024-bit modulus N. To enable 1024-bit modular arithmetic in $\mathbb{Z}_N^*$, we developed a Solidity library which may be of independent interest. This size of N corresponds to approximately $\lambda = 80$ bits of security. Although this security level is no longer deemed cryptographically safe, the secrecy of the HTLP solutions is only guaranteed up to time T regardless, so this security level will suffice for our use case as long as the best-known factoring attack takes at least T time. A 2012 estimate for factoring 1024-bit integers is about a year [BHL12], which is significantly longer than the typical submission period of a decentralized auction or election.

The main factors influencing gas cost (see **??**) are submission size, correctness proof size, and verification complexity. These factors mainly depend on the packing parameter $\ell \in [m]$, which determines a storage-computation trade-off with the following extremes:

One aggregate HTLP for all. If $\ell = m$, the contract maintains a single aggregate HTLP Z. This greatly reduces the on-chain space requirements of the resulting voting or auction scheme at the expense of typically more complex and larger submission correctness proofs.

[21] Open-sourced at https://github.com/a16z/cicada.

One aggregate HTLP per candidate. If $\ell = 1$, the contract must maintain m aggregate HTLPs $\{Z_j\}_{j\in[m]}$. This increases the on-chain storage, but the submissions of correctness proofs become smaller and cheaper to verify.

In **??**, we empirically explore this trade-off space by measuring the gas costs of various deployments of our framework with a range of parameter settings M, n, m, w, ℓ.

First, we briefly describe the proof systems used for each scheme we implement; detailed descriptions are given in **??**.

Binary voting. In a binary vote (i.e., approval voting with $m = 1$), such as a simple yes/no referendum, users prove that the submitted ballot $Z = (u, v)$ is an exponential ElGamal HTLP with solution 0 or 1: $(u = g^r \wedge v = h^r) \vee (u = g^r \wedge vy^{-1} = h^r)$. This is achieved via OR-composition [CDS94] of two sigma protocols for discrete logarithm equality [CP93].

Cumulative voting. In cumulative voting, each user distributes w votes among m candidates. To accommodate a larger number of candidates, our implementation keeps m tally HTLPs Z_j, one for each candidate (in other words, $\ell = 1$). Each voter i submits m ballots $Z_{ij} = (g^{r_{ij}}, h^{r_{ij}} y^{s_{ij}})$ for all $j \in [m]$. Besides proving (using the protocol zk-PoKS) that each HTLP is well-formed (the same r_{ij} is used in both terms), the voter must prove that $0 \leq s_{ij}\ \forall\ j \in [m]$ and $\sum_{j=1}^{m} s_{ij} = w$. The first condition is shown with a proof of positive solution (zk-PoPS) via Legendre's three-square decomposition theorem [Gro05]. As a building block, we use a proof of square solution (zk-PoKSqS) to show that a puzzle solution is a square. The second condition is proven by providing the randomness $R_i = \prod_j r_{ij}$ which opens $\prod_j Z_{ij}$ to w.

Sealed-bid auction. To illustrate two extremes of the packing spectrum, we implement two flavors of sealed-bid auctions. The first uses a single aggregate HTLP as described in **??** (this can be viewed as $\ell = b$, where $b = \lceil \log_2(M) \rceil$ is the bit-length of a bid): Bidder i submits a single HTLP $Z_i = (g^{r_i}, h^{r_i} y^{s_i})$, proving well-formedness with zk-PoKS and two zk-PoPS to show $0 \leq s \leq M$. The coordinator aggregates the ith bidder's bid by adding $M^{i-1} \cdot Z_i$ to its tally.

The second approach applies b aggregate HTLPs (i.e., $\ell = 1$): Each bidder i submits b HTLPs $\{Z_{ij}\}_{j\in[b]}$ and uses the same proof system as in binary voting to prove their well-formedness, i.e., the user inserted for each bit of the bid 0 or 1. The coordinator adds $2^i \cdot Z_{ij}$ to each corresponding aggregate HTLP Z_j.

Submission costs. The on-chain cost of submitting a bid/ballot is the cost of running the Aggr function by the contract, i.e., the verification of the well-formedness proofs plus adding the users' submissions to the tally HTLPs (if and only if they verify). We report our measurements without packing (i.e., $\ell = 1$) in Table 5. Submitting a binary vote ballot costs 418,358 gas ($\approx$ 11.02 USD on

Binary vote		**Cumulative vote** ($\ell = 1$)				
m	1	2	3	4	5	6
Aggr	418,358	3,391,514	5,081,542	6,781,389	8,489,786	10,208,185
Finalize	115,690	269,505	397,789	521,895	644,814	770,934
Sealed-bid auction ($\ell = b$)		**Sealed-bid auction** ($\ell = 1$)				
b	any	8	10	12	14	16
Aggr	3,055,107	3,586,022	4,488,050	5,394,047	6,304,164	7,218,905
Finalize	147,634	1,005,208	1,253,119	1,497,760	1,749,489	2,003,282

Table 5: Gas costs for Cicada cumulative voting and sealed-bid auctions with various numbers of candidates m, bid bit-lengths b (max. bid $M = 2^{b-1}$), and packing parameters ℓ.

Ethereum).[22] For cumulative voting, the submission cost scales linearly in m: with $m = 2$ candidates, submitting a ballot costs 3,391,514 gas ($\approx$ 94.49 USD), and each additional candidate adds $\approx$ 1,699,847 gas ($\approx$ 44.79 USD).

An auction with a single HTLP for each bit of the bid (the $\ell = 1$ case) requires a submission cost of 3,586,022 gas ($\approx$ 94.49 USD) for an 8-bit bid. Every additional bit in the submitted bid burns $\approx$ 451,014 gas ($\approx$ 11.89 USD). On the other hand, if one applies packing, i.e., $\ell = b$, then the cost of submitting a sealed bid is constant at 3,055,107 gas ($\approx$ 80.50 USD). As seen in Table 5, with bid-space $M = 2^7$ it is already more economical to have a single aggregate HTLP and use a packing scheme, despite more complex bid-correctness proofs.

Finalization costs. Our voting and auction schemes end with solving the tally HTLP(s) off-chain, i.e., computing $(g^r)^{2^T} (= h^r)$. With exponential ElGamal, solving the puzzle also requires a brute-force discrete logarithm computation by the off-chain solver. The correctness of this computation is proven to the contract with Wesolowski's PoE [Wes19] (recalled in **??**). The Finalize cost comes from verifying the PoE(s) on-chain, which burns 101,066 gas ($\approx$ 2.66 USD) per proof. Without packing, the untrusted solver must provide a Wesolowski proof per tally HTLP, so the Finalize gas cost is linear in the number of tally HTLPs, as evidenced by Table 5. A portion of the Wesolowki verification cost comes from checking that the challenge is a prime number. In our implementation, the prover provides a Baillie-PSW [PSW80] primality certificate, whose verification cost is 44,972 gas ($\approx$ 1.18 USD).

Verification costs. We implemented the sigma protocols described in **??** in Solidity and report their verification costs in Table 6. With Groth's trick [Gro05] in the proof of positivity (zk-PoPS), we must decompose the integer solution into

[22] We can estimate gas costs for approval voting using the cost of binary voting, as the former uses a disjunction of m copies of the same NIZK and thus scales linearly.

the sum of only three squares. Therefore, the gas cost of verifying zk-PoPS equals the cost of verifying three proofs of knowledge of square solutions (zk-PoKSqS) and one proof of knowledge of equal solution (zk-PoSEq).

Sigma protocol	**Verification gas cost**
Proof of Exponentiation (PoE [Wes19])	101,066
PoK of solution (zk-PoKS)	266,096
Proof of solution equality (zk-PoSEq)	336,155
Proof of square solution(zk-PoKSqS)	336,168
Proof of positive solution (zk-PoPS)	1,351,958

Table 6: EVM gas costs of verification for the proof systems described in **??**.

In the short-term, deploying on Layer 2 (L2) already brings these costs down by 1–2 orders of magnitude. For example, when deploying our implementation on the Optimism L2 rollup, casting a binary vote would cost less than 0.30 USD. Further optimizations (e.g., Karatsuba multiplication [KO62], batched Wesolowski proof verification [Rot21], or verification via efficient zkSNARKs [Gro16, GWC19]) can bring the costs down even more.

4.10 Extensions

We introduce extensions to the Cicada framework that may be useful in future applications.

4.10.1 Everlasting ballot privacy for HTLP-based protocols

Noemi: Note this introduces interactivity but off-chain

The basic Cicada framework does not guarantee long-term ballot privacy. Submissions are public after the Open stage. This is because users publish their HTLPs on-chain: once public, the votes contained in the HTLPs are only guaranteed to be hidden for the time it takes to compute T sequential steps, after which point it is plausible that someone has computed the solution. In many applications, it is desirable that individual ballots remain hidden *even after voting has ended* since the lack of everlasting privacy may facilitate coercion and vote-buying. As mentioned in Section 4.5, this can be achieved modularly by first decoupling the ballots from their voters via a privacy-enhancing overlay. Alternatively, we describe how the Seal procedure can be modified to prevent the opening of individual ballots, achieving everlasting privacy.

Observe that all known *efficient HTLP constructions* are of the form $(u, v) = (g^r, h'^r X)$,[23] where the solution is encoded in X and recovering it requires

[23] In the exponential ElGamal case, $h' = h$, while in the Paillier construction, $h' = h^N$ (see Section 2.3). We will drop the tickmark on h' in the remainder of this section to avoid notational clutter.

Off-chain batching

Public parameters: A semiprime N and $h, y \in \mathbb{Z}_N^*$, a voting scheme $\Sigma : \mathcal{X}^n \to \mathcal{Y}$.

Let $P_1, \ldots, P_k$ be a group of $k < n$ parties with addresses $\mathsf{addr}_1, \ldots, \mathsf{addr}_k$ wishing to batch their ballots $(u_i, v_i) := (g^{r_i}, h^{r_i} X_i)$.

1. Each party broadcasts v_i. Now, every party can compute $v := \prod_i v_i = h^R X$, which encodes the sum of their submissions.
2. The parties use an $k-1$ malicious-secure MPC protocol [DKL+13, Kel20] on inputs u_i to compute $u := \prod_i u_i = g^R$.
3. They also compute two distributed-prover zero-knowledge proofs [DPP+22] in the MPC: (i) a discrete logarithm equality proof π_R that $\mathsf{dlog}_g(u) = \mathsf{dlog}_h(v)$ with distributed witness R, and (ii) a submission correctness proof π_s that the aggregated solution s encoded in X is consistent with the sum of k valid submissions, i.e., $s \in k \cdot \mathcal{X}$. Let $\pi_{\mathsf{batch}} = (\pi_R, \pi_s)$.
4. Finally, each party signs the final aggregated submission $Z_{\mathsf{batch}} = (u, v)$.

Output: $(Z_{\mathsf{batch}}, \pi_{\mathsf{batch}}, \{\mathsf{addr}_1, \ldots, \mathsf{addr}_k\}, \{\sigma_1, \ldots, \sigma_k\})$.

On-chain batched ballot submission

Public parameters: Cicada public parameters pp.

1. The designated party P_1 submits $Z_{\mathsf{batch}}, \pi_{\mathsf{batch}}, \{\mathsf{addr}_1, \ldots, \mathsf{addr}_k\}$, $\{\sigma_1, \ldots, \sigma_k\}$ to the tallying contract, which verifies the proofs and signatures, and adds (u, v) to the tally HTLP Z as in the basic protocol.
2. If P_1 doesn't submit by time $T - \tau$, any other party in the batch group can submit instead.

Figure 12: The on- and off-chain ballot batching protocols that $k < n$ parties can use to achieve everlasting ballot privacy.

recomputing $h^r = (g^r)^{2^T}$ via repeated squaring of the first component. Our insight is that the puzzle information-theoretically hides the solution X without the first component. Importantly, publishing g^r is not necessary in any of our HTLP-based voting protocols *except as a means to verifiably compute the first component of the final HTLP*, i.e., $g^R = g^{\sum_{i\in[n]} r_i}$. The observation that g^R can be computed *without* revealing the individual values g^{r_i} enables us to construct the first practical and private voting protocols that guarantee *everlasting* ballot privacy with a single on-chain round.

For simplicity, consider a protocol in which both the ballot of user i and the tally consists of a single HTLP, respectively $Z_i = (g^{r_i}, h^{r_i} X_i)$ and $Z = (g^R, h^R X)$. Observe that for everlasting ballot privacy, updates to Z must inherently be batched: a singleton update $\mathsf{Aggr}(\mathsf{pp}, Z, Z_i, \pi) \to (g^{R+r_i}, h^{R+r_i} Y)$ (for some Y) would reveal $g^{r_i} = g^{R+r_i}/g^R$, which is the opening information to Z_i, as the quotient of the first component of Z after and before the update. Hence, the ballot X_i of user i would be recoverable in T sequential steps, i.e., after computing $h^{r_i} = (g^{r_i})^{2^T}$.

Batching ballot submissions off-chain in groups of k allows parties to achieve everlasting privacy as long as at least one party is honest. The parties aggregate their submissions off-chain as $(g^R, h^R X) = (\prod_i g^{r_i}, \prod_i h^{r_i} X_i)$ and compute a proof π_{batch} of well-formedness in a distributed-prover zero-knowledge proof protocol [DPP+22]. We use the observation that the individual second components v_i are hiding to optimize the batching by computing $h^R X$ in the clear (see Figure 12 for details).

Borda ballot correctness

todo: incomplete

1. $\mathcal{P}$ sends $(g^a \bmod N, (1+N)^a \bmod N^2)$ for $a \leftarrow\!\!\$\ \mathbb{Z}_N$.
2. $\mathcal{V}$ sends a challenge $e \leftarrow\!\!\$\ [1, n]$.
3. $\mathcal{P}$ sends $(g^{r\cdot e+a}, h^{r\cdot N\cdot e+a}(1+N)^{s\cdot e+a} \bmod N^2)$. $\mathcal{V}$ accepts iff the following hold:

This idea opens up a new design space for the MPC protocol used for batching, such as doing the randomness generation in a preprocessing phase instead, allowing dynamic additions to the anonymity set, optimizing the batch proof generation, and dealing with parties who fail to submit. We leave the full exploration of this large design space and related questions to future work.

4.10.2 Succinct ballot-correctness proofs

Real-world elections often have hundreds of candidates, e.g., Optimism's retroactive public good funding [Opt]. However, the state-of-the-art ballot correctness proofs [BBC+23, Gro05] for all voting schemes (e.g., FPTP, approval voting,

Preprocessing ballots for succinct ballot-correctness proofs

Public parameters: The common reference string $\mathsf{crs} := \{g_1^{\tau^j}\}_{j=1}^{d} \in \mathbb{G}_1^d$. A semiprime N and $h, y \in \mathbb{Z}_N^*$. A voting scheme $\Sigma : \mathcal{X}^n \to \mathcal{Y}$, e.g., approval voting.

1. Let $\mathcal{X}$ be the set of correct ballots and $|\mathcal{X}| = d$
2. Let $f(x) \in \mathbb{F}_p^{\leq d}[x]$ be a univariate polynomial s.t. $\forall s_i \in \mathcal{X} : f(i) := \mathsf{Pack}(s_i)$. The polynomial $f(x)$ could be computed using Lagrangian interpolation.
3. Let com be the KZG commitment to the polynomial f.

Output: com.

Figure 13: Preprocessing ballots to enable succinct ballot-correctness proofs.

etc.) are linear in the number of candidates, rendering these schemes impractical in the blockchain setting. To counter these issues, we design constant-size ballot correctness proofs with constant verification time at the expense of an added preprocessing phase. The high-level idea is as follows. All correct ballots (e.g., $\{\mathsf{Pack}(s) : s \in \{0,1\}^m\}$ in the case of approval voting) are inserted into an accumulator or polynomial commitment (PC) [KZG10] during a transparent preprocessing phase. When users submit their votes $Z \leftarrow\!\$\ \mathsf{HTLP.Gen}(s)$, they prove in zero-knowledge that Z encodes a correct ballot, i.e., the users show that the solution s of Z had been previously inserted into the accumulator or PC with a succinct (blinded) membership proof [ZBK+22].

In this section, we assume that a common reference string for the KZG polynomial commitment (PC) scheme [KZG10] is already available to users, namely $\mathsf{crs} := \{g_1^{\tau^j}\}_{j=1}^{d}$, where $g_1 \in \mathbb{G}_1$ is a generator in a bilinear pairing-friendly cyclic group $\mathbb{G}_1$ over $\mathbb{F}_p$ for some prime p, $\tau \leftarrow\!\$\ \mathbb{F}_p$ hidden to everyone. The crs is typically established during a sequential, secure multi-party computation (MPC), e.g., [BGG19].

Let us assume that users have established during a preprocessing phase (Figure 13) a short commitment com that encodes all the possible ballots in a particular voting scheme, e.g., $\mathcal{X} = [0,1]^m$ for approval voting. The size of classical proofs of well-formedness, e.g., OR-composition of sigma-protocols, scale linearly in the number of candidates m. The following proof strategy yields a constant-size proof of correctness for moderately-sized $\mathcal{X}$, i.e., $|\mathcal{X}| \leq d$[24].

First, given a ballot $Z = (g^r, h^r y^s) \in \tilde{\mathbb{G}}_1 \times \tilde{\mathbb{G}}_2$, the user creates an elliptic curve point $Z_1 = h_1^r y_1^s \in \mathbb{G}_1$ for random generators $h_1, y_1 \leftarrow\!\$\ \mathbb{G}_1$ in a pairing-

[24]The largest KZG CRS we know of [Prib] is for $d = 2^{28}$, so in the case of $\mathcal{X} = [0,1]^m$ this strategy requires $m \leq 28$.

friendly group. Using the discrete logarithm across different groups techniques developed in [COPZ22], the user can show that Z and Z_1 have the same discrete logarithms r and s with for their bases $h, y \in \mathbb{G}, h_1, y_1 \in \mathbb{G}_1$, respectively. Now that Z_1 and the polynomial commitment are in the same pairing-friendly group $\mathbb{G}_1$, the user can create a blinded KZG opening proof [ZBK+22] to prove ballot correctness. Specifically, the proof π shows that the value s in Z_1 matches an evaluation of the polynomial f committed by com at some (hidden) point j, i.e., $f(j) = s$. Note that the verifier only sees constant-size commitments of f, j, and s. Since the blinded KZG proof π is also constant-size, this strategy yields the first succinct ballot-correctness proofs for many common voting schemes, e.g., approval and range voting.

4.10.3 Coercion-Resistance

Lastly, we briefly outline how one could add coercion-resistance [JCJ05] to our framework. In the e-voting literature, there are two main pathways to obtaining coercion-resistance: receipt-freeness or allowing unlikable revotes. Receipt-freeness seems challenging to achieve in the blockchain context, and we leave it to future work. Therefore, we follow the revoting paradigm akin to Lueks et al. [LQT20]. One can allow indistinguishable revotes as follows. We could store our ballots in a zero-knowledge set (e.g., Semaphore is a readily available implementation of this concept for Ethereum [Eth19]). Additionally, we would need to store a Merkle tree of nullifiers on-chain containing the ballots that are revoked due to revoting. Whenever users want to revote, they could prove in zero-knowledge that they revoke a previous ballot that they inserted in the zero-knowledge set while they reveal the accompanying nullifier and insert it into the nullifier tree. We leave it to future work to flesh out the technical details and implementation of such an extension.

4.10.4 Bayesian truth serum

Bayesian truth serum [Pre04] is a method for eliciting truthful subjective answers where objective truth does not exist or is not knowable. The core of the idea is to reward answers that are "surprisingly common" by leveraging respondents' own predictions of what will be common. Thus, for a question with many (mutually exclusive) potential answers, the score of user i responding $\mathbf{x}_i := (x_{i1}, \ldots, x_{im})$ and $\mathbf{y}_i := (y_{i1}, \ldots, y_{im})$ is calculated as

$$\mathsf{score}_i := \sum_{j \in [m]} x_{ij} \log \frac{\overline{x}_j}{\overline{y}_j} + \alpha \sum_{j \in [m]} \overline{x}_j \log \frac{y_{ij}}{\overline{x}_j} \tag{7}$$

where $\alpha > 0$ is a constant. The variable $x_{ij} \in \{0, 1\}$ denotes user i's decision (choose or don't choose) for option $j \in [m]$, $\overline{x}_j$ is the empirical frequency of choice j over all the users' answers, y_{ij} is user i's estimate of $\overline{x}_j$ (i.e., their estimate of the probability of answer j among all users), and $\overline{y}_j$ is the empirical (geometric) average of y_{ij} over all the users' answers. Since each user can only

choose a single answer, x_{ij} will be 0 for all but one value of j, which we denote j^*. Thus, we can think of the equation above as equivalent to

$$x_{ij^*} \log \frac{\overline{x}_{j^*}}{\overline{y}_{j^*}} + \alpha \sum_{j \in [m]} \overline{x}_j \log \frac{y_{ij}}{\overline{x}_j}.$$

The first term is referred to as the *information score* and the second as the *prediction score*. The information score is highest when the user's choice k^* is "surprisingly common", i.e., when the empirical frequency of answer j^* ($\overline{x}_{j^*}$) is higher than the crowd's estimate of the empirical frequency of j^* ($\overline{y}_{j^*}$). Therefore participants are incentivized to submit their truthful responses, even (and especially) if they believe them to be uncommon.

The prediction score is the Kullback-Leibler divergence [KL51] between the user's estimate of the average answer and the true average answer, weighted by α. This is maximized when the two values are equal (i.e., the divergence is 0), and so incentivizes truthful reporting of y_{ij}, the user's estimate of $\overline{x}_j$.

We show how Bayesian truth serum can be implemented in the Cicada framework. First, rewrite Equation (7) as

$$\mathsf{score}_i := \sum_{j \in [m]} x_{ij}(\log \overline{x}_j - \overline{y}'_j) + \alpha \sum_{j \in [m]} \overline{x}_j (y'_{ij} - \log \overline{x}_j) \tag{8}$$

where $y'_{ij} = \log y_{ij}$ and $\overline{y}'_j = \log \overline{y}_j$. The smart contract will use two (lists of) HTLPs $\mathcal{Z}^{\mathsf{tally}}_{\overline{\mathbf{x}}}, \mathcal{Z}^{\mathsf{tally}}_{\overline{\mathbf{y}}'}$ to keep track of two running "tallies":

$$\overline{\mathbf{x}} = (\overline{x}_1, \ldots, \overline{x}_m) = \sum_i \mathbf{x}_i$$

$$\overline{\mathbf{y}}' = (\overline{y}'_1, \ldots, \overline{y}'_m) = \sum_i \frac{1}{n} \mathbf{y}'_i$$

Each user's ballot consists of the vectors $\mathbf{x}_i, \mathbf{y}'_i$, where $\mathbf{x}_i \in [0,1]^m$ has ℓ_1 norm 1 and $\mathbf{y}'_i = \log \mathbf{y} \in \mathbb{N}^m$ with $\sum_{j \in [m]} y_{ij} = n$. Assuming no packing for simplicity, the ballot is encoded as three lists of HTLPs: a list of linear HTLPs $\mathcal{Z}^{\mathsf{ans}}_i := \{Z^{\mathsf{ans}}_{ij}\}_{j \in [m]}$ for the entries of $\mathbf{x}_i$, and two lists of (respectively) linear and multiplicative HTLPs $\mathcal{Z}^{+}_i := \{Z^{+}_{ij}\}_{j \in [m]}$ and $\mathcal{Z}^{\times}_i := \{Z^{\times}_{ij}\}_{j \in [m]}$, both encoding the entries of $\mathbf{y}'_i$. The smart contract coordinator must ensure that the following hold:

Check #1a. All Z^{ans}_{ij} encode $x_{ij} \in [0,1]$.

Check #1b. $\sum_{j \in [m]} x_{ij} = 1$.

Check #2a. All Z^{+}_{ij} encode $y'_{ij} > 0$.

Check #2b. $\sum_{j \in [m]} 2^{y'_{ij}} = n$ (assuming log base 2).

Check #3. $Z^{\times}_{ij}$ contains the same value as Z^{+}_{ij} for all $j \in [m]$.

Most of these checks can be achieved using the protocols in Section 4.7: Check #1a with the binary solution protocol, #1b and #2b by providing randomness which opens the homomorphic sum to the correct value, and #2a with zk-PoPS. Check #2b additionally requires a zero-knowledge proof of exponentiation, e.g., [BBF19]. Because the puzzles to check in #3 use different constructions, we can't apply zk-PoSEq directly; instead, one can combine two zk-PoKS proofs with a standard PoK for discrete logarithm.

The aggregation algorithm $\mathsf{Aggr}((\mathcal{Z}^{\mathsf{tally}}_{\overline{\mathbf{x}}}, \mathcal{Z}^{\mathsf{tally}}_{\overline{\mathbf{y}}'}), i, \mathcal{Z}_i, \pi_i)$ updates the tally to $(\mathcal{Z}^{\mathsf{tally}}_{\overline{\mathbf{x}}} \boxplus \mathcal{Z}^{\mathsf{ans}}_i, \mathcal{Z}^{\mathsf{tally}}_{\overline{\mathbf{y}}'} \boxplus \frac{1}{n} \cdot \mathcal{Z}^{+}_i)$. During the opening phase, anyone can solve for the final tallies $\overline{\mathbf{x}}_{\mathsf{final}}, \overline{\mathbf{y}}'_{\mathsf{final}}$ and the individual user submissions $\{(\mathbf{x}_i, \mathbf{y}'_i)\}_{i\in[n]}$. If correct, they are used in Finalize to compute the final set of scores as follows:

1. Let $\overline{\mathbf{x}}' := \log \overline{\mathbf{x}}$. Compute $\mathbf{I}' := \overline{\mathbf{x}}' - \overline{\mathbf{y}}'$ and $\mathbf{P}' := \overline{\mathbf{x}} \cdot \overline{\mathbf{x}}'$.
2. For each user $i \in [n]$:
   (a) Compute i's information score $I_i := \sum_{j\in[m]} I_{ij}$, where $\mathbf{I}_i = (I_{i1}, \ldots, I_{im}) := \mathbf{x}_i \cdot \mathbf{I}'$.
   (b) Compute i's prediction score $P_i := \sum_{j\in[m]} P_{ij}$, where $\mathbf{P}_i = (P_{i1}, \ldots, P_{im}) := \overline{\mathbf{x}} \cdot \mathbf{y}'_i - \mathbf{P}'$.
   (c) User i's score is $I_i - P_i$.

4.10.5 A trusted setup protocol for the CJSS scheme

Chvojka, Jager, Slamanig, and Striecks [CJSS21] describe how to combine a public-key encryption scheme with a TLP to obtain a private voting or auction protocols which, unlike the HTLP-based approach suggested by [MT19], is "solve one, get many for free". The high-level idea of the protocol is to encrypt each user's bid with a common public key whose corresponding secret key is inserted into a TLP (see Figure 14). Therefore, none of the bids can be decrypted until the corresponding encryption secret key is obtained by solving the TLP. One drawback of this scheme, however, is that it requires an additional trusted setup procedure to create a TLP containing the secret key corresponding to the encryption public key used. Furthermore, unlike the HTLP approach, the setup cannot be reused and must be re-run for every protocol invocation.

We observe that, for encryption schemes with discrete-log key-pairs such as Cramer-Shoup [CS98], there is a natural decentralized setup protocol secure against all-but-one corruptions. Using the blockchain as a broadcast channel (similar to [NRBB24]), a simple sequential MPC protocol to set up the parameters works as follows. Suppose there is some smart contract that stores the public key $\mathsf{pk} = g^{\mathsf{sk}} \mod N$ and a TLP Z_{sk} containing sk (initially, one can set $\mathsf{sk} = 0$). Each contributor i can update pk by adding s_i homomorphically in the exponent and contributing an HTLP $Z_i = (g^{r_i} \mod N, h^{r_i \cdot N} \cdot (1+N)^{s_i})$. The contribution must be accompanied by a proof of well-formedness. For the previous state $\mathsf{pk}, Z_{\mathsf{sk}}$, contributor i proves that its contribution pk_i, Z_i passes the following checks:

The CJSS Framework

Let Π_{E} be a CCA-secure public-key encryption scheme, TLP a time-lock puzzle scheme, and $\Sigma : \mathcal{X}^n \to \mathcal{Y}$ a base voting/auction protocol.

$\mathsf{Setup}(1^\lambda, T) \xrightarrow{\$} (\mathsf{pp}, \mathcal{Z})$**.** Sample a key-pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \Pi_{\mathsf{E}}.\mathsf{Gen}(1^\lambda)$ and TLP parameters $\mathsf{pp}_{\mathsf{tlp}} \leftarrow \mathsf{TLP}.\mathsf{Setup}(1^\lambda, T)$. Compute $Z_{\mathsf{sk}} \leftarrow\!\!\$\ \mathsf{TLP}.\mathsf{Gen}(\mathsf{pp}_{\mathsf{tlp}}, \mathsf{sk})$ and return $\mathsf{pp} := (\mathsf{pp}_{\mathsf{tlp}}, \mathsf{pk})$ and $\mathcal{Z} := (Z_{\mathsf{sk}}, \bot)$.

$\mathsf{Seal}(\mathsf{pp}, i, s) \xrightarrow{\$} (\mathsf{ct}_i, \pi_i)$**.** Parse pk from pp and compute an encrypted bid/ballot as $\mathsf{ct}_i \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{pk}, s_i)$ along with a proof π_i that ct_i is a valid encryption under pk.

$\mathsf{Aggr}(\mathsf{pp}, \mathcal{Z}, \mathsf{ct}_i, \pi_i) \to \mathcal{Z}'$**.** Verify π_i. If the check passes, parse $\mathcal{Z} := (Z_{\mathsf{sk}}, \mathcal{C})$ and update to $\mathcal{Z}' := (Z_{\mathsf{sk}}, \mathcal{C} \cup \{\mathsf{ct}_i\})$.

$\mathsf{Open}(\mathsf{pp}, \mathcal{Z}, \bot) \to \mathsf{sk}$**.** Let $\mathcal{Z} := (Z_{\mathsf{sk}}, \mathcal{C})$ and publish $\mathsf{sk} \leftarrow \mathsf{HTLP}.\mathsf{Solve}(\mathsf{pp}_{\mathsf{tlp}}, Z_{\mathsf{sk}})$.

$\mathsf{Finalize}(\mathsf{pp}, \mathsf{sk}) \to y$**.** Use the secret key sk to decrypt each ciphertext $\mathsf{ct}_i \in \mathcal{C}$ to $s_i \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}_i)$. Compute the final result in the clear as $\Sigma(s_1, \ldots, s_n)$.

Figure 14: The "solve one, get many for free" paradigm (CJSS) [CJSS21].

Check #1**.** It knows the discrete logarithm of pk_i with respect to the base g. This can be achieved with a proof of knowledge of the exponent [Sch90].

Check #2**.** It knows the representation of the HTLP contribution Z_i with respect to the bases $g, h^N, (1+N)$ (i.e., the discrete logarithms r_i, r_i, s_i). This can be proven by a "knowledge of representation" proof system in groups of unknown order (e.g., [BBF19]).

Check #3**.** Finally, the discrete logarithms a, b, c from check #2 are such that $a = b$ and $c = \mathsf{dlog}_g(\mathsf{pk}_i)$.

The state is updated with the ith contribution iff all the checks pass. After the update, $Z_{\mathsf{sk}} := Z_{\mathsf{sk}} \cdot Z_i$ and $\mathsf{pk} := \mathsf{pk} \cdot \mathsf{pk}_i = g^{s+s_i}$. A single honest contributor suffices to guarantee a uniformly distributed keypair.

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 [NNN+23, Mak23].

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 [But13, 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 [Fil], transactions on the Chia Network [Chi24], and a BLS smart contract wallet [Pri22]. With the event of account abstraction on Ethereum [Eth24a], 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[25] 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

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

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 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

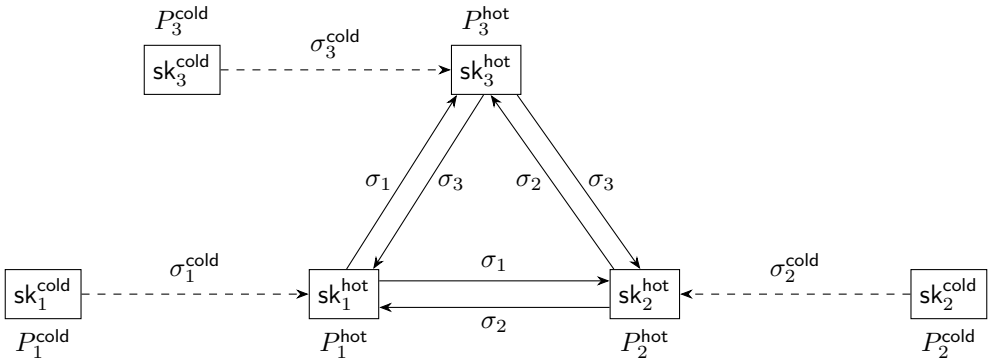

Figure 15: A hot-cold threshold wallet with $n = 3$. Given a message m, each P_i^{cold} uses its cold share $\mathsf{sk}_i^{\text{cold}}$ to compute a cold partial signature σ_i^{cold} on m and sends it to its hot storage. P_i^{hot} uses m and its hot share $\mathsf{sk}_i^{\text{hot}}$ to compute σ_i^{hot}, 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.

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 $I_i = (P_i^{\text{hot}}, P_i^{\text{cold}})$. 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 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 15 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 11 (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* $\mathsf{dk} \leftarrow\!\!\$\, \mathbb{Z}_p$*. 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 11 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 $(\mathsf{vk}, \mathsf{sk}) := (g^r, r)$.[26] The result is that each

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

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 2.10) holds on random inputs. In order for the output of $\mathcal{H}$ to have sufficient entropy to 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 2.10, 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 [Tha23] 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 $s \leftarrow\!\$\ \mathbb{Z}_p$ 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

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

(where the 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

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 $r \leftarrow\!\!\$\ \mathbb{Z}_p$, 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 2.9).
2. Sample $s \leftarrow\!\!\$\ \mathbb{Z}_p$ 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 16: 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.

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.

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 5.7.3, we show how to avoid trusting C in the refresh stage.

The final hot proof of remembrance is summarized as Π_{EKS} in Figure 16.

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 2.7.1) 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 2.7.1 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 17, 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 5.7.3, 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 18) 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 19 and 20. 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 $x \leftarrow\!\$\, \mathbb{Z}_p \setminus \{0\}$. 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 17: The encrypted secret sharing functionality $\mathcal{F}_{\mathsf{SS}}$.

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

  1. Sample $\mathsf{dk} \leftarrow\!\!\$\ \mathbb{Z}_p$ 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 18: 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 the cold storage. Each P_i^{cold} only stores a single decryption key $\mathsf{dk}_i \in \mathbb{G}_2^2$,

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 2.7.1 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 19: 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 20: Our BLS hot-cold threshold wallet protocol (proofs of remembrance and proactive refresh).

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[27], which we summarize in Section 2.11.

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., $(\texttt{sid}, \mathsf{Action}, P_{\mathsf{sender}}, (\mathsf{contents}))$ or $(\texttt{sid}, \mathsf{ActionDone}, P_{\mathsf{recipient}}, \mathsf{publicvars}, (\mathsf{contents}))$.

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 21, 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 22.

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.

[27] Specifically, we use the version presented in [CLOS02].

[28] 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 $\mathsf{dk}_i \leftarrow\!\$\, \mathbb{Z}_p$ 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 $t \le n$ 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 $y := \perp$.
  2. Otherwise, sample $x \leftarrow\!\$\, \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
  3. Generate t-out-of-n Shamir Shares of x as $x_1, \ldots x_n \in \mathbb{Z}_p$. Let $y_i := g_2^{x_i}\ \forall i \in [n]$.
  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 21: The BLS hot-cold threshold wallet functionality $\mathcal{F}_{\mathsf{HC}}$ (registration).

Signing

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

Proofs of Remembrance

- On input $(\mathsf{sid}, \mathsf{CProof}, P_i^{\mathsf{cold}}, (C))$, retrieve $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathsf{dk}_i, *, *, \mathtt{corrupted}, *) \in L_{\mathsf{cold}}$. If $\mathtt{corrupted} = 1$, send $(\mathsf{sid}, \mathsf{CProofRequest}, \mathcal{A}, (P_i^{\mathsf{cold}}))$ 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 $(\mathsf{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$.
- On input $(\mathsf{sid}, \mathsf{HProof}, C, (P_i^{\mathsf{hot}}))$, retrieve $(P_i^{\mathsf{hot}}, C, \tilde{x}_i, \pi_i, *, *, *, \mathtt{corrupted}, *) \in L_{\mathsf{hot}}$ and $(C, *, *, *, \mathsf{com}) \in D$. If $\mathtt{corrupted} = 1$, send $(\mathsf{sid}, \mathsf{HProofRequest}, \mathcal{A}, (P_i^{\mathsf{hot}}))$ 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 $(\mathsf{sid}, \mathsf{HProofResult}, C, (P_i^{\mathsf{hot}}, b))$.

Proactive Refresh

- On input $(\mathsf{sid}, \mathsf{ShareRefresh}, C)$, it proceeds as follows:
  1. Output $\perp$ if a prior share refresh from C is still pending.[28]
  2. Retrieve $(C, \mathcal{I}, t, y, \mathsf{com}, \mathtt{time}) \in D$ for the maximum value of $\mathtt{time}$.
  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 23 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 6 (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 `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 `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 `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 `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 `corrupted` to 1. ($\mathcal{A}$ receives P_i^{cold}'s tapes and $\mathcal{E}$ is notified.)

Figure 23: The BLS hot-cold threshold wallet functionality $\mathcal{F}_{\mathsf{HC}}$ (adversarial interfaces).

– Otherwise, if $\tilde{P}_i^{\mathsf{cold}}$ is honest, it stores $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, \mathtt{corrupt} = 0) \in L'_{\mathsf{cold}}$, 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 $(\mathtt{sid}, \mathsf{ClientRegister}, C, (t, \mathcal{I}))$ on $\tilde{C}$'s outgoing communication tape to $\mathcal{F}_{\mathsf{HC}}$ once $\mathcal{A}$ delivers $(\mathtt{sid}, \mathsf{SSSetup}, C, (t, \{P_i^{\mathsf{hot}}\}_{i\in[n]}, *)$ from C to $\mathcal{F}_{\mathsf{SS}}$ in the internal interaction. $\mathcal{F}_{\mathsf{HC}}$ will send $(\mathtt{sid}, \mathsf{ClientRegistered}, C, (\mathsf{vk}))$ to $\tilde{C}$ and $(\mathtt{sid}, \mathsf{ClientRegistered}, P_i^{\mathsf{hot}}, (C, b_i))$ 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 $(\mathtt{sid}, \mathsf{TSign}, P_i^{\mathsf{hot}}, (C, m))$, 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 $(\mathtt{sid}, \mathsf{TSignResult}, P_i^{\mathsf{hot}}, (C, m, \sigma_i))$.

– 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 $(\mathtt{sid}, \mathsf{CProof}, C, (P_i^{\mathsf{cold}}))$, 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 $(\mathtt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$ 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 $(\mathtt{sid}, \mathsf{CProofResult}, C, (P_i^{\mathsf{cold}}, b))$ to $\tilde{C}$ once $\mathcal{A}$ delivers the message from $\mathcal{F}_{\mathsf{PK}}$ to C in the internal execution.

On input $(\mathtt{sid}, \mathsf{HProof}, P_i^{\mathsf{hot}}, (C))$, $\mathcal{S}$ 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}. □

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.[29] 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 7. Hot share generation ($\mathsf{ClientRegister}$), producing hot

[29] 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 Π_{Ref}.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 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 8. 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 9.

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

Table 7: Benchmarks of our algorithms regardless of threshold

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

Table 8: 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 9. The ShareRefresh times for C and P_i^{hot} include the UCVrfy resp. HVrfy times of the refresh proofs in Table 9.

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

Table 9: Sizes for our various proof systems regardless of threshold

5.7.3 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 $t - 1 \leq d$ 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 24 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 25). 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 26, where changes with respect to locally running the ideal functionality $\mathcal{F}_{\mathsf{SS}}$ are shown in blue.

Theorem 7. *The encrypted secret sharing protocol in Figure 26 UC-realizes $\mathcal{F}_{\mathsf{SS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model.*

Proof. todo: □

5.8 Hot-Cold Threshold Schnorr

Next, we show how to adapt FROST [KG20, CKM21, BCK+22], a two-round threshold Schnorr protocol, to the hot/cold setting. (Specifically, we use the optimized version, FROST2, introduced in [CKM21].) Noemi: Justify why FROST (very popular in practice, non-interactive signing phase, security with aborts/concurrency?)

The idea of our protocol, given in Figure 27, is to let the cold parties generate their pair's nonce commitment (D_i, E_i) so that only they know the corresponding discrete logarithms d_i, e_i. This way, given a partial cold signature (where the cold secret is blinded by a function of d_i, e_i), a corrupt hot storage is unable to recover the cold secret which would allow it to forge an unlimited number of partial signatures.

Like FROST, our protocol uses two hash functions $H_1, H_2 : \{0,1\}^* \to \mathbb{Z}_q^*$. During the message-dependent phase, the hot parties help with computing the nonce commitment R by aggregating every other pair's nonce commitment except their own (we call this incomplete commitment R'). The final contribution to R is added by the cold party itself to ensure that the hot does not replace it with a nonce commitment for which it knows the discrete logarithms. Then it also computes the pair's contribution to the second signature component, namely z, by using the secret nonce values d_i, e_i. A blinded version of this component is sent to the hot storage as the cold partial signature. The hot party can use its secret s_i^{hot} to unblind it and simultaneously add the pair's secret key share sk_i, thereby obtaining z_i.

Finally, like in the classic FROST protocol, the hot parties broadcast their values z_i and reconstruct the full signature as $\sigma := (R, \sum_{i \in SS} z_i) = (g^k, k + c \cdot \mathsf{sk})$.

Remark 1. *The above protocol can be modified in a straightforward manner to FROST1 threshold signatures, the original version of FROST introduced in [KG20].*

5.8.1 Security Analysis

We first recall the proof of security for the original FROST2 protocol [BCK+22].

Theorem 8 (informal)**.** *FROST2 is unforgeable assuming the one-more discrete logarithm (OMDL) assumption holds.*

Proof (informal). Given an adversary $\mathcal{A}$ against the unforgeability of FROST2 (specifically, against TS-SUF-2 [BCK+22]), we will construct an adversary $\mathcal{B}$ for the OMDL game. Specifically, given $2q_s + t$ OMDL challenges $(A_0, \ldots, A_{t-1}, U_1, V_1, \ldots, U_{q_s}, V_{q_s})$, $\mathcal{B}$ will output the corresponding discrete logarithms $(a_0, \ldots, a_{t-1}, u_1, v_1, \ldots, u_{q_s}, v_{q_s})$ using only $2q_s + t - 1$ queries to the discrete logarithm oracle $\textsc{Dlog}(\cdot)$.

The idea is as follows: $\mathcal{B}$ will run $\mathcal{A}$, programming the FROST instance with the a_j as coefficients of the polynomial f used to share the secret key (that is, $\mathsf{vk} := A_0 = g^{a_0}$ and $\mathsf{vk}_i := \prod_{0 \le j \le t-1} A_j^{i^j} = g^{f(i)} = g^{\mathsf{sk}_i}$) and using the (U_k, V_k) as preprocessing tokens for FROST partial signatures. Note that this already requires some $\textsc{Dlog}(\cdot)$ queries by $\mathcal{B}$: $\mathcal{A}$ receives as input the secret keys of the corrupted parties (the set CS), computed as $\mathsf{sk}_i := \textsc{Dlog}(\mathsf{vk}_i) \forall i \in CS$. Computing a partial signature for party i and signing set SS using the token (U_k, V_k) also requires a single query to compute $z_i := \textsc{Dlog}(U_k V_k^d \mathsf{vk}_i^{c\lambda_i^{SS}})$, where d, c are random oracle outputs.

Once $\mathcal{A}$ outputs its forgery (m^*, σ^*), where $\sigma^* = (R^*, z^*)$ and R^* was output by the Ith (partial) signing oracle query which also used some preprocessing

token (U_{k^*}, V_{k^*}), we rewind and program different RO outputs for that signing oracle query (and all the following ones as well). By the generalized Forking Lemma [BN06], $\mathcal{A}$ will still output a forgery on the same message m^* with a reasonably high probability and now we have two "versions" (forks) of signing oracle queries/responses *with the same preprocessing tokens* (U_k, V_k). We can use this information to extract all of the discrete logarithms with sufficiently few $\text{DLOG}(\cdot)$ queries.

We will use a tick mark to denote the corresponding variable in the second fork, e.g., σ^* is the forgery output in the first fork and $\sigma^{*\prime}$ in the second. Note that $\sigma^* = (R^*, z^*)$ and in our simulated FROST2 these will be computed as $R^* = \prod_{i \in SS} R_i S_i^{\rho}$ (where each $(R_i, S_i) = (U_k, V_k)$ for some k) and $z^* = \sum_{i \in SS} \text{DLOG}(R_i S_i^d \mathsf{vk}_i^{c\lambda_i^{SS}})$, where ρ, c are random oracle outputs. In particular, at exactly the forking point, the second fork will use the same first RO output (so $\rho' = \rho$ and thus $R^{*\prime} = R^*$) but a different second RO output (so $c' \neq c$ and thus $z^{*\prime} \neq z^*$).

In more detail, to compute $\text{DLOG}(A_i)$ for all $i = 0, \ldots, t-1$, $\mathcal{B}$ will do the following:

- We have $g^{z^*} = (\prod_i g^{R_i S_i^{\rho}}) \cdot \mathsf{vk}^c = R^* A_0^c$ and similarly $g^{z^{*\prime}} = R^* A_0^{c'}$. So, $g^{z^* - z^{*\prime}} = g^{a_0(c-c')}$ and we can compute $a_0 = \frac{z^* - z^{*\prime}}{c - c'}$.
- Next, we can look at all the different (honest) parties for which $\mathcal{A}$ made a partial signing oracle query for m^* (and the same signing set SS[30]). By the winning condition, $\mathcal{A}$ can only have made queries for $< t - |CS|$ honest parties (call this set Q^*). For each of those queries, in our simulation we computed z_i^* by querying $\text{DLOG}(U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}})$ for some k. So $g^{z_i^*} = U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}}$ and in the second fork $g^{z_i^{*\prime}} = U_k V_k^{\rho} \mathsf{vk}_i^{c'\lambda_i^{SS}}$ and therefore $g^{z_i^* - z_i^{*\prime}} = g^{\mathsf{sk}_i(c-c')}$ Noemi: their paper left out λ_i^{SS} in the exponent. Now we can solve for $\mathsf{sk}_i = f(i) = \frac{z_i^* - z_i^{*\prime}}{c - c'}$ Noemi: should be divided by λ_i^{SS}.
- At this point, including the values of sk_i queried as inputs to $\mathcal{A}$, we have $|Q^*| + |CS|$ points $f(i)$ and also $a_0 = f(0)$. We will pick some set of indices Q' of size $t - |Q^*| - |CS| - 1$ and query $\text{DLOG}(\mathsf{vk}_j)$ for all $j \in Q'$ to get $\mathsf{sk}_j = f(j)$. Now we have t evaluations of f, so we interpolate $f(X)$ and evaluate at $X = 1, \ldots, t-1$ to get $a_1, \ldots, a_{t-1}$.

Now we will look more systematically at all the partial signature queries/responses we simulated (at all times beyond just the forking point) to recover $\text{DLOG}(U_j), \text{DLOG}(V_j)$ for all $j = 1, \ldots, q_s$. Remember that we issued each (U_j, V_j) as a preprocessing token, and $\mathcal{A}$ may or may not have later requested a partial signature using that token. To simplify notation, let $\mathsf{sk}_i^+ := \mathsf{sk}_i \cdot \lambda_i^{SS}$.

Case 0: $\mathcal{A}$ never requested a signature for that (U_j, V_j) (in either fork). In this case we have no information about u_j, v_j and must request them from

[30] for the case of TS-SUF-i with $i = 2$.

the oracle: $u_j = \text{DLOG}(U_j)$, $v_j = \text{DLOG}(V_j)$. Note however that this means we didn't have to simulate the partial signature for this token, so we saved a $\text{DLOG}(\cdot)$ query there.

Case 1: $\mathcal{A}$ only requested a signature for that (U_j, V_j) in *one* fork. Because we simulated this partial signature in one fork, we know a value $z_i = \text{DLOG}(U_j V_j^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}})$. We can compute $v_j = \text{DLOG}(V_j)$ and $u_j = z_i - \rho \cdot v_j - c \cdot \mathsf{sk}_i^+$.

Case 2: $\mathcal{A}$ requested a signature for that (U_j, V_j) in *both* forks. Because we simulated partial signatures in both forks, we know values z_i, z_i' such that $g^{z_i} = U_j V_j^d \mathsf{vk}_i^{c\lambda_i^{SS}}$ and $g^{z_i'} = U_j V_j^{d'} \mathsf{vk}_{i'}^{c'\lambda_{i'}^{SS}}$. (Note the preprocessing token (U_j, V_j) could have been assigned to different parties $i \neq i'$ in the two forks.) We now have various cases based on whether this request happened before, at, or after the forking point.

- **Case 2a: *after* the forking point.** In this case, $d \neq d'$ and $c \neq c'$. So $g^{z_i - z_i'} = g^{v_j(d-d') + \mathsf{sk}_i^+ c - \mathsf{sk}_{i'}^+ c'}$. Therefore we compute $v_j = \frac{z_i - c \cdot f(i)\lambda_i^{SS} - z_i' + c' \cdot f(i')\lambda_{i'}^{SS}}{d - d'}$ and $u_j = z_i - d v_j - c \cdot f(i)\lambda_i^{SS}$.
- **Case 2b: *at* the forking point.** In this case, $d = d'$ and $c \neq c'$ (see above where we calculated the a_i's). Here we can learn nothing about u_j, v_j since their terms are the same in both forks, so we have to resort to the same strategy as Case 1 and compute $v_j = \text{DLOG}(V_j)$ and $u_j = z_i - d v_j - c \cdot \mathsf{sk}_i^+$.
- **Case 2c: *before* the forking point.** In this case, $d = d'$ and $c = c'$ and again we have no information about u_j, v_j and compute $v_j = \text{DLOG}(V_j)$ and $u_j = z_i - d v_j - c \cdot \mathsf{sk}_i^+$ as in Case 1 (and 2b). Note that for simulating the partial signature, we were able to reuse the outputs from the first fork in the second, so as in Case 0 we save a $\text{DLOG}(\cdot)$ query.

Clearly we have now calculated all $2q_s + t$ discrete logarithms correctly, so it remains to show that we only used $2q_s + t - 1$ $\text{DLOG}(\cdot)$ queries to do it:

- $|CS|$ to compute the inputs $\{\mathsf{sk}_i\}_{i \in CS}$ to $\mathcal{A}$,
- $|Q'|$ to compute $a_0, \ldots, a_{t-1}$,
- 2 total for each $j = 1, \ldots, q_s$ to compute u_j, v_j and/or to simulate the partial signature for U_j, V_j—except for Case 2b, where we have to simulate partial signatures (each for some party i) for U_j, V_j in both forks *at* the forking point (i.e., on the winning message m^*). Remember that these were requested for parties in the set Q^*, so we only have to make $|Q^*| < t - |CS|$ (by the winning condition) queries $\text{DLOG}(V_j)$ (i.e., only $|Q^*|$ values of j fall under Case 2b).

In total, the number of $\text{DLog}(\cdot)$ queries is $|CS| + |Q'| + 2q_s + |Q^*|$, and since $|Q'| = t - |Q^*| - |CS| - 1$ and $|CS| < t$ (by the winning condition of unforgeability) this is equal to $2q_s + t - 1$ queries.

Finally, we need to argue about the advantage of $\mathcal{B}$ assuming $\mathcal{A}$ has non-negligible advantage. There are two events (called BadPPO and $\mathsf{BadHash}$ in [BCK+22]) in which $\mathcal{B}$ will fail even though $\mathcal{A}$ wins (resp. either by failing to simulate FROST to $\mathcal{A}$ or by having colliding RO outputs when forking/rewinding $\mathcal{A}$). Let $\mathcal{C}$ denote the subroutine of $\mathcal{B}$ dedicated to simulating FROST ($\mathcal{C}$ "accepts", i.e., $\text{acc}(\mathcal{C}) = 1$, if it does so successfully and $\mathcal{A}$ wins):

$$\Pr[\text{acc}(\mathcal{C})] \geq \Pr[\mathsf{Adv}(\mathcal{A})] - \Pr[\mathsf{BadPPO}]$$

By a variant of the Forking Lemma [BCK+22, Lemma 5.3],

$$\Pr[\mathsf{Adv}(\mathcal{B})] \geq \text{acc}(\mathcal{C})^2/q - \Pr[\mathsf{BadHash}]$$

where q is the number of RO queries in $\mathcal{C}$ (equal to $q_s + q_h + 1$, where q_s is the number of partial signature queries and q_h is the number of any other RO queries $\mathcal{A}$ makes). Noemi: idk where +1 comes from

By the bounds on $\Pr[\mathsf{BadPPO}]$ and $\Pr[\mathsf{BadHash}]$ (see [BCK+22]), we end up with

$$\Pr[\mathsf{Adv}(\mathcal{B})] \geq \Pr[\mathsf{Adv}(\mathcal{A})]^2/q - 3q^2/p.$$

So the existence of an adversary $\mathcal{A}$ with non-negligible advantage in the TS-SUF-2 game implies an adversary $\mathcal{B}$ with non-negligible advantage in the OMDL game, which contradicts our assumption. □

We now show how to modify the above proof to show that the protocol in Figure 27 is HC-SUF-2-secure. For readability, in Figure 28 we give a version of the unforgeability oracles which is tailored to HC-FROST2.

Theorem 9. *The hot/cold variant of FROST2 described in* **??** *is unforgeable assuming OMDL.*

Proof (sketch). The idea is for the reduction to program the OMDL challenges into the inputs to $\mathcal{A}$ like before. So, the $\mathsf{vk}, \mathsf{vk}_1, \ldots, \mathsf{vk}_n$ will encode the challenges $A_0, \ldots, A_{t-1}$. For $\mathcal{A}$'s inputs, for $i \in CS_{\mathsf{hot}} \cap CS_{\mathsf{cold}}$, the reduction samples $\Delta_i \leftarrow\!\!\$\ \mathbb{Z}_q$, computes $\mathsf{sk}_i \leftarrow \text{DLog}(\mathsf{vk}_i)$, and sets $s_i^{\mathsf{cold}} = \Delta_i$, $s_i^{\mathsf{hot}} = \mathsf{sk}_i + \Delta_i$; for $i \in CS_{\mathsf{hot}} \setminus CS_{\mathsf{cold}}$, it lets $s_i^{\mathsf{hot}} \leftarrow\!\!\$\ \mathbb{Z}_q$, and similarly for $i \in CS_{\mathsf{cold}} \setminus CS_{\mathsf{hot}}$, it lets $s_i^{\mathsf{cold}} \leftarrow\!\!\$\ \mathbb{Z}_q$.

The reduction will answer PSignO queries as before using the challenges (U_k, V_k). It also handles RO queries like before. Previously, partial signature queries were answered by querying the discrete log oracle once to get $z_i \leftarrow \text{DLog}(U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}})$. Now we have to instead handle queries to CSignO and HSignO, but we will do it again with a single DLog query *per pair*. In $\text{CSignO}(i, \cdot)$ (only answered if P_i^{cold} is honest), if P_i^{hot} is honest we sample $r \leftarrow\!\!\$\ \mathbb{Z}_q$ and compute $z_i^{\mathsf{cold}} = \text{DLog}(U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}}) - c \cdot r_i \lambda_i^{SS}$, otherwise $z_i^{\mathsf{cold}} = \text{DLog}(U_k V_k^{\rho}) - c \cdot s_i^{\mathsf{hot}} \lambda_i^{SS}$. For $\text{HSignO}(i, \sigma_i^{\mathsf{cold}})$ (similarly only answered

if P_i^{hot} is honest), if P_i^{cold} is honest we compute $z_i = \text{DLOG}(U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}})$, where we can reuse the output of a potential previous DLOG query in CSIGNO on the same value. Otherwise $z_i = \text{DLOG}(\mathsf{vk}_i^{c\lambda_i^{SS}}) + \sigma_i^{\mathsf{cold}} + c\Delta_i\lambda_i^{SS}$.

Given a successful forgery by $\mathcal{A}$, computing the discrete logarithms to the OMDL challenges is similar. As before, $a_0 = \frac{z^* - z^{*\prime}}{c - c'}$. Then, for every query to either CSIGNO or HSIGNO we can recover a point $f(i)$ as follows:

case 1: $\mathcal{A}$ queried HSIGNO$(i, \cdot)$ and $i \in HS_{\mathsf{cold}}$. That means we have two values z_i, z_i' from the two forks such that $g^{z_i} = U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}}$ and $g^{z_i'} = U_k V_k^{\rho} \mathsf{vk}_i^{c'\lambda_i^{SS}}$ and therefore $g^{z_i - z_i'} = g^{\mathsf{sk}_i \lambda_i^{SS}(c-c')}$. Then $\mathsf{sk}_i = f(i) = \frac{z_i - z_i'}{\lambda_i^{SS}(c-c')}$.

case 2: $\mathcal{A}$ queried HSIGNO$(i, \cdot)$ and $i \notin HS_{\mathsf{cold}}$. Then the response z_i was instead computed as $z_i = \text{DLOG}(\mathsf{vk}_i^{c\lambda_i^{SS}}) + z_i^{\mathsf{cold}} + c\Delta_i\lambda_i^{SS}$. Thus, todo: assuming (forking lemma?) that $\mathcal{A}$ queried on the same z_i^{cold} from the two forks we have values z_i, z_i' such that $g^{z_i - z_i'} = g^{(\mathsf{sk}_i + \Delta_i)\lambda_i^{SS}(c-c')}$. Then $\mathsf{sk}_i = f(i) = \frac{z_i - z_i'}{\lambda_i^{SS}(c-c')} - \Delta_i$, where the value Δ_i was sampled as an input for $\mathcal{A}$ so it is known to the reduction.

case 3: $\mathcal{A}$ queried CSIGNO$(i, \cdot)$ and $i \notin HS_{\mathsf{hot}}$. Here the response is computed as $z_i^{\mathsf{cold}} = \text{DLOG}(U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}}) - c \cdot s_i^{\mathsf{hot}} \lambda_i^{SS}$. Given the responses from the two forks, we have $z_i^{\mathsf{cold}}, z_i^{\mathsf{cold}\prime}$ such that $g^{z_i^{\mathsf{cold}} - z_i^{\mathsf{cold}\prime}} = g^{(\mathsf{sk}_i - s_i^{\mathsf{hot}})\lambda_i^{SS}(c-c')}$ and can compute $\mathsf{sk}_i = f(i) = \frac{z_i^{\mathsf{cold}} - z_i^{\mathsf{cold}\prime}}{\lambda_i^{SS}(c-c')} + s_i^{\mathsf{hot}}$, where the value s_i^{hot} was sampled as an input for $\mathcal{A}$ so it is known to the reduction.

case 4: $\mathcal{A}$ queried CSIGNO$(i, \cdot)$ and $i \in HS_{HS}$, and $\mathcal{A}$ did not query HSIGNO$(i, \cdot)$. (Note that if $\mathcal{A}$ *did* query HSIGNO$(i, \cdot)$, this falls under case 1 instead.) Now we have values $z_i^{\mathsf{cold}} = \text{DLOG}(U_k V_k^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}}) - c \cdot r_i \lambda_i^{SS}$ and $z_i^{\mathsf{cold}\prime} = \text{DLOG}(U_k V_k^{\rho} \mathsf{vk}_i^{c'\lambda_i^{SS}}) - c' r_i \lambda_i^{SS}$, where we use the same random coins for r_i in both forks. Therefore we compute $\mathsf{sk}_i = f(i) = \frac{z_i^{\mathsf{cold}} - z_i^{\mathsf{cold}\prime}}{(c-c')\lambda_i^{SS}} + r_i$.

Recall that these queries are captured by the set $Q^* = (S_{\mathsf{hot}}(lr) \cap CS_{\mathsf{cold}}) \cup (CS_{\mathsf{hot}} \cap S_{\mathsf{cold}}(lr)) \cup (S_{\mathsf{hot}}(lr) \cap S_{\mathsf{cold}}(lr))$. Along with $a_0 = f(0)$ and the values of sk_i we queried as inputs to $\mathcal{A}$, we now have $|Q^*| + |CS_{\mathsf{hot}} \cap CS_{\mathsf{cold}}| + 1$ evaluations of the polynomial f. As in the previous proof, next we pick a set Q' of indices, where $|Q'| = t - |Q^*| - 1$, and query $\mathsf{sk}_j = f(j) = \text{DLOG}(\mathsf{vk}_j)$ for $j \in Q'$. This gives us a total of t evaluations of f, allowing us to recover all the remaining values $a_1, \ldots, a_{t-1}$.

To compute the discrete logarithms of the challenges $(U_1, V_1), \ldots, (U_{q_s}, V_{q_s})$, we can use the same approach as before, unmodified. Anytime $\mathcal{A}$ requested a signature using some token (U_j, V_j) todo: except HSIGNO with corrupt cold, need to figure that out..., we know a value $z_i = \text{DLOG}(U_j V_j^{\rho} \mathsf{vk}_i^{c\lambda_i^{SS}})$ and the same case analysis applies.

It remains to count the DLog$(\cdot)$ queries used:

- $|CS_{\mathsf{hot}} \cap CS_{\mathsf{cold}}|$ for $\mathcal{A}$'s inputs,
- $|Q'|$ to compute $a_0, \ldots, a_{t-1}$, and
- $2q_s + |Q^*|$ to compute u_j, v_j and/or simulate the partial signature(s) for U_j, V_j.

Remember that $|Q'| = t - |Q^*| - |CS_{\mathsf{hot}} \cap CS_{\mathsf{cold}}| - 1$, so once again the total number of DLog queries used by the reduction is $2q_s + t - 1$.

□

6 Conclusion

We described three examples of how advanced cryptography can improve the security and trust assumptions of blockchain applications in a practical and deployable manner: naysayer proofs, the Cicada voting and auction framework, and hot-cold threshold wallets. While these works have had significant industry discussion and/or collaboration, it is my hope that they will also see practical deployment in the near future.

The ideas discussed in this dissertation also have significant potential to spark additional discussion and future work. The naysayer paradigm can be substituted for classic zero-knowledge proofs in any application which uses the latter as a building block. A more theoretical analysis of their complexity, lower bounds, and application to particular classes of underlying verifier circuits may also be fruitful. There is also room for optimizing naysayer provers to particular underlying proof systems. Finally, like optimistic rollups, naysayer proofs would benefit from a rigorous analysis of how to set collateral and delay periods in optimistic settings.

We have shown that the Cicada framework is a useful blueprint for fair on-chain elections and auctions and have shown its practicality for many popular protocols. Extending Cicada to other, non-additive protocols could be a useful future work. Additionally, each of the proposed extensions can be explored in greater detail as well.A

Finally, our hot-cold threshold wallet protocols cover two popular threshold signature schemes used in the blockchain ecosystem today, but it may be desirable or necessary to extend other threshold signatures to this setting. Furthermore, although our protocols already offer several advanced functionalities, deployments may wish to extend our protocols more, e.g., by adding distributed key generation, hot share refreshes, and secret resharing.

Cryptography has seen significant deployment in the blockchain ecosystem and driven some of its most important innovations. As the space continues to grow, cryptographic protocols and primitives have a large role to play in furthering security, decentralization, and scalability.

References

[Aba23] Aydin Abadi. Decentralised repeated modular squaring service revisited: Attack and mitigation. Cryptology ePrint Archive, Paper 2023/1347, 2023. https://eprint.iacr.org/2023/1347. *[p. 41]*

[ABB+22] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+: Submission to the nist post-quantum project, v.3.1. http://sphincs.org/data/sphincs+-r3.1-specification.pdf, June 2022. *[p. 30]*

[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. *[pp. 67 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. *[p. 31]*

[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. *[pp. 67 and 12]*

[AHK17] Shahzad Asif, Md Selim Hossain, and Yinan Kong. High-throughput multi-key elliptic curve cryptosystem based on residue number system. *IET Computers & Digital Techniques*, 11(5):165–172, 2017. *[p. 39]*

[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. *[pp. 68 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. *[pp. 16, 9, and 10]*

[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. *[pp. 68 and 14]*

[AOZZ15] Joël Alwen, Rafail Ostrovsky, Hong-Sheng Zhou, and Vassilis Zikas. Incoercible multi-party computation and universally composable receipt-free voting. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 763–780. Springer, Heidelberg, August 2015. *[p. 35]*

[azt] Aztec documentation. https://docs.aztec.network/. *[p. 20]*

[BBC+23] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Arithmetic sketching. In Helena Handschuh and Anna Lysyanskaya,

editors, *CRYPTO 2023, Part I*, volume 14081 of *LNCS*, pages 171–202. Springer, Heidelberg, August 2023. *[p. 61]*

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019. *[pp. 40, 49, 52, 53, 65, and 66]*

[BBHR18a] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl, July 2018. *[p. 29]*

[BBHR18b] 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. *[p. 29]*

[BCK10] Endre Bangerter, Jan Camenisch, and Stephan Krenn. Efficiency limitations for S-protocols for group homomorphisms. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 553–571. Springer, Heidelberg, February 2010. *[p. 49]*

[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. *[pp. 68, 89, 90, 93, and 14]*

[BCM05] Endre Bangerter, Jan Camenisch, and Ueli Maurer. Efficient proofs of knowledge of discrete logarithms and representations in groups with hidden order. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 154–171. Springer, Heidelberg, January 2005. *[p. 40]*

[BDJ+06] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 142–147. Springer, Heidelberg, February / March 2006. *[p. 35]*

[BDM06] Jean-Claude Bajard, Sylvain Duquesne, and Nicolas Méloni. *Combining Montgomery Ladder for Elliptic Curves Defined over $\mathbb{F}_p$ and RNS Representation*. PhD thesis, LIR, 2006. *[p. 39]*

[BEHG20] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Advances in Financial Technologies*, 2020. *[p. 31]*

[BF01] Dan Boneh and Matthew Franklin. Efficient generation of shared RSA keys. *Journal of the ACM (JACM)*, 48(4):702–722, 2001. *[p. 44]*

[BFL+22] Vitalik Buterin, Dankrad Feist, Diederik Loerakker, George Kadianakis, Matt Garnett, Mofi Taiwo, and Ansgar Dietrichs. Eip-4844: Shard blob transactions. https://eips.ethereum.org/EIPS/eip-4844, Feb 2022. *[p. 33]*

[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. *[p. 31]*

[BGG19] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *FC 2018 Workshops*, volume 10958 of *LNCS*, pages 64–77. Springer, Heidelberg, March 2019. *[p. 62]*

[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. *[pp. 30 and 31]*

[BHL12] Daniel J Bernstein, Nadia Heninger, and Tanja Lange. Facthacks: Rsa factorization in the real world. https://www.hyperelliptic.org/tanja/vortraege/facthacks-29C3.pdf, 12 2012. *[p. 56]*

[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. *[pp. 68 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. *[pp. 13, 14, 68, and 7]*

[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. *[pp. 67, 70, 12, and 16]*

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006. *[p. 91]*

[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. *[pp. 68 and 14]*

[BT94] Josh Cohen Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *26th ACM STOC*, pages 544–553. ACM Press, May 1994. *[p. 38]*

[BTZ22] Mihir Bellare, Stefano Tessaro, and Chenzhi Zhu. Stronger security for non-interactive threshold signatures: BLS and FROST. Cryptology ePrint Archive, Report 2022/833, 2022. https://eprint.iacr.org/2022/833. *[p. 14]*

[Buc] Chris Buckland. Fraud proofs and virtual machines. https://medium.com/@cpbuckland88/fraud-proofs-and-virtual-machines-2826a3412099. *[p. 22]*

[But13] Vitalik Buterin. Deterministic wallets, their advantages and their understated flaws. https://bitcoinmagazine.com/technical/deterministic-wallets-advantages-flaw-1385450276, 11 2013. *[p. 67]*

[But14] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *White paper*, 2014. *[pp. 6 and 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. *[pp. 17 and 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. *[pp. 68 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. *[pp. 68 and 14]*

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 174–187. Springer, Heidelberg, August 1994. *[pp. 50 and 57]*

[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. *[p. 33]*

[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. *[pp. 68 and 14]*

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. A homomorphic LWE based E-voting scheme. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 245–265. Springer, Heidelberg, 2016. *[pp. 35 and 36]*

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 103–118. Springer, Heidelberg, May 1997. *[p. 39]*

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), 1981. *[p. 31]*

[CHI+21] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with

a dishonest majority. In *2021 IEEE Symposium on Security and Privacy*, pages 590–607. IEEE Computer Society Press, May 2021. *[p. 44]*

[Chi24] Chia Network. BLS keys. https://docs.chia.net/bls-keys/, 2024. *[p. 68]*

[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. *[pp. 89 and 33]*

[CJSS21] Peter Chvojka, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Versatile and sustainable timed-release encryption and sequential time-lock puzzles (extended abstract). In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021, Part II*, volume 12973 of *LNCS*, pages 64–85. Springer, Heidelberg, October 2021. *[pp. 35, 65, and 66]*

[CKM21] Elizabeth Crites, Chelsea Komlo, and Mary Maller. How to prove schnorr assuming schnorr: Security of multi- and threshold signatures. Cryptology ePrint Archive, Report 2021/1375, 2021. https://eprint.iacr.org/2021/1375. *[p. 89]*

[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. *[pp. 17, 81, 10, and 26]*

[COPZ22] Melissa Chase, Michele Orrù, Trevor Perrin, and Greg Zaverucha. Proofs of discrete logarithm equality across groups. Cryptology ePrint Archive, Report 2022/1593, 2022. https://eprint.iacr.org/2022/1593. *[p. 63]*

[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. *[pp. 32, 40, 50, and 57]*

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25. Springer, Heidelberg, August 1998. *[p. 65]*

[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. *[pp. 67 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. *[pp. 67 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. *[pp. 68 and 14]*

[DK02] Ivan Damgård and Maciej Koprowski. Generic lower bounds for root extraction and signature schemes in general groups. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 256–271. Springer, Heidelberg, April / May 2002. *[p. 40]*

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013. *[p. 60]*

[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. *[p. 30]*

[DKL+21] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1). https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf, Feb 2021. *[p. 30]*

[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. *[pp. 68 and 14]*

[dLNS17] Rafaël del Pino, Vadim Lyubashevsky, Gregory Neven, and Gregor Seiler. Practical quantum-safe voting from lattices. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1565–1581. ACM Press, October / November 2017. *[pp. 35 and 36]*

[DM10] Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 183–200. Springer, Heidelberg, February 2010. *[p. 44]*

[DPP+22] Pankaj Dayama, Arpita Patra, Protik Paul, Nitin Singh, and Dhinakaran Vinayagamurthy. How to prove any NP statement jointly? Efficient distributed-prover zero-knowledge protocols. *PoPETs*, 2022(2):517–556, April 2022. *[pp. 60 and 61]*

[dyd] dYdX. https://dydx.exchange/. *[p. 20]*

[Edg23] Ben Edgington. *Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond*. 2023. https://eth2book.info/latest/. *[pp. 68 and 14]*

[EL04] Edith Elkind and Helger Lipmaa. Interleaving cryptography and mechanism design: The case of online auctions. In Ari Juels, editor, *FC 2004*, volume 3110 of *LNCS*, pages 117–131. Springer, Heidelberg, February 2004. *[p. 35]*

[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. *[pp. 72 and 17]*

[Eme13] Peter Emerson. The original Borda count and partial voting. *Social Choice and Welfare*, 40:353–358, 2013. *[pp. 42 and 46]*

[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. *[pp. 67 and 12]*

[Eth19] Ethereum Foundation. Semaphore: a zero-knowledge set implementation for ethereum. https://github.com/semaphore-protocol/semaphore, May 2019. *[pp. 45 and 63]*

[Eth23a] Ethereum. Account Abstraction. ethereum.org/en/roadmap/account-abstraction, 2023. *[p. 30]*

[Eth23b] Ethereum. Optimistic rollups. https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/, 2023. *[p. 21]*

[Eth24a] Ethereum. Account abstraction. https://ethereum.org/en/roadmap/account-abstraction/, march 2024. *[p. 68]*

[Eth24b] Ethereum. Merkle patricia trie. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/, jul 2024. *[p. 28]*

[Eya21] Ittay Eyal. On cryptocurrency wallet design. Cryptology ePrint Archive, Report 2021/1522, 2021. https://eprint.iacr.org/2021/1522. *[pp. 67 and 12]*

[FG14] Jon Fraenkel and Bernard Grofman. The Borda Count and its real-world alternatives: Comparing scoring rules in Nauru and Slovenia. *Australian Journal of Political Science*, 49(2), 2014. *[p. 43]*

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru, specification v1.2. https://falcon-sign.info/falcon.pdf, Jan 2020. *[p. 30]*

[Fic] Kelvin Fichter. Why is the optimistic rollup challenge period 7 days? https://kelvinfichter.com/pages/thoughts/challenge-periods/. *[p. 22]*

[Fil] Filecoin. Filecoin spec. https://spec.filecoin.io/algorithms/crypto/signatures/. *[p. 68]*

[FK97] Uriel Feige and Joe Kilian. Making games short (extended abstract). In *29th ACM STOC*, pages 506–516. ACM Press, May 1997. *[p. 21]*

[FMW22] Robin Fritsch, Marino Müller, and Roger Wattenhofer. Analyzing voting power in decentralized governance: Who controls DAOs? *arXiv preprint arXiv:2204.01176*, 2022. *[p. 35]*

[FOO93] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology—AUSCRYPT'92: Workshop on the Theory and Application of Cryptographic Techniques Gold Coast, Queensland, Australia, December 13–16, 1992 Proceedings 3*, pages 244–251. Springer, 1993. *[p. 35]*

[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. *[pp. 16, 53, 73, 9, and 18]*

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009. *[p. 35]*

[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. *[pp. 68 and 14]*

[GG22] Aayush Gupta and Kobi Gurkan. Plume: An ecdsa nullifier scheme for unique pseudonymity within zero knowledge proofs. Cryptology ePrint Archive, Paper 2022/1255, 2022. https://eprint.iacr.org/2022/1255. *[p. 35]*

[GGJ+24] Sanjam Garg, Noemi Glaeser, Abhishek Jain, Michael Lodder, and Hart Montgomery. Hot-cold threshold wallets with proofs of remembrance, 2024. Available at https://eprint.iacr.org/2023/1473. *[p. 66]*

[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. *[pp. 68 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. *[p. 19]*

[Gir20] Damien Giry. Cryptographic key length recommendation. https://www.keylength.com/, May 2020. *[p. 30]*

[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. *[pp. 68 and 14]*

[Gol01] Oded Goldreich. *Foundations of Cryptography*, volume 1. Cambridge University Press, 2001. *[p. 8]*

[Gro05] Jens Groth. Non-interactive zero-knowledge arguments for voting. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 467–482. Springer, Heidelberg, June 2005. *[pp. 11, 40, 50, 54, 57, 58, and 61]*

[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. *[pp. 26 and 59]*

[GSZB24] Noemi Glaeser, István András Seres, Michael Zhu, and Joseph Bonneau. Cicada: A framework for private non-interactive on-chain auctions and voting. In *Workshop on Cryptographic Tools for Blockchains*, 2024. *[p. 35]*

[GTN11] Mahadevan Gomathisankaran, Akhilesh Tyagi, and Kamesh Namuduri. Horns: A homomorphic encryption scheme for cloud computing using residue number system. In *2011 45th Annual Conference on Information Sciences and Systems*, pages 1–5, 2011. *[p. 39]*

[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. *[pp. 28 and 59]*

[GY19] Hisham S Galal and Amr M Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *2nd Workshop on Trusted Smart Contracts*, pages 265–278. Springer, 2019. *[p. 35]*

[HS00] Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 539–556. Springer, Heidelberg, May 2000. *[p. 11]*

[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. *[pp. 67 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. *[pp. 16, 9, and 10]*

[JCJ05] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In Vijay Atluri, Sabrina De Capitani di Vimercati, and Roger Dingledine, editors, *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005*, pages 61–70. ACM, 2005. *[p. 63]*

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020. *[p. 60]*

[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. *[pp. 14, 68, 89, and 90]*

[KGC+18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018. *[pp. 21 and 24]*

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951. *[p. 64]*

[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. *[pp. 67, 69, 12, and 14]*

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk*, 145(2):293–294, 1962. *[p. 59]*

[KPT+22] Igor Anatolyevich Kalmykov, Vladimir Petrovich Pashintsev, Kamil Talyatovich Tyncherov, Aleksandr Anatolyevich Olenev, and Nikita Konstantinovich Chistousov. Error-correction coding using polynomial residue number system. *Applied Sciences*, 12(7):3365, 2022. *[p. 39]*

[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. *[pp. 15, 33, 62, and 8]*

[Lab23] Offchain Labs. Inside Arbitrum Nitro. https://docs.arbitrum.io/inside-arbitrum-nitro/, 2023. *[p. 21]*

[Lin22] Yehuda Lindell. Simple three-round multiparty schnorr signing with full simulatability. Cryptology ePrint Archive, Report 2022/374, 2022. https://eprint.iacr.org/2022/374. *[p. 14]*

[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. *[pp. 68 and 14]*

[LQT20] Wouter Lueks, Iñigo Querejeta-Azurmendi, and Carmela Troncoso. VoteAgain: A scalable coercion-resistant voting system. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 1553–1570. USENIX Association, August 2020. *[p. 63]*

[LW18] Steven P Lalley and E Glen Weyl. Quadratic voting: How mechanism design can radicalize democracy. In *AEA Papers and Proceedings*, volume 108, 2018. *[pp. 43 and 52]*

[Mak23] Nikolaos Makriyannis. Practical key-extraction attacks in leading MPC wallets. *MPTS 2023: NIST Workshop on Multi-party Threshold Schemes 2023*, 2023. *[p. 67]*

[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. *[pp. 28 and 33]*

[MT19] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 620–649. Springer, Heidelberg, August 2019. *[pp. 8, 9, 10, 35, 39, and 65]*

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008. *[p. 6]*

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 116–125. ACM Press, November 2001. *[p. 45]*

[NNN+23] Duy Hieu Nguyen, Anh Khoa Nguyen, Huu Giap Nguyen, Thanh Nguyen, and Anh Quynh Nguyen. New key extraction attacks on threshold ecdsa implementations. https://www.verichains.io/tsshock/verichains-tsshock-wp-v1.0.pdf, August 2023. *[p. 67]*

[NRBB24] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. In Christina Pöpper and Lejla Batina, editors, *ACNS 24, Part III*, volume 14585 of *LNCS*, pages 105–134. Springer, Heidelberg, March 2024. *[p. 65]*

[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. *[pp. 16 and 9]*

[Ope23] OpeanSea. How to sell an NFT, June 2023. *[p. 35]*

[Opt] Optimism. Optimism RetroPGF 2: Badgeholder manual. Accessed 2023-09-18. *[p. 61]*

[Opt23a] Optimism. Rollup Protocol. https://community.optimism.io/docs/protocol/2-rollup-protocol, 2023. *[p. 21]*

[Opt23b] Optimism. What is the Optimism Collective?, February 2023. *[p. 35]*

[PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017. *[p. 24]*

[PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016. *[p. 24]*

[PE23] Privacy and Scaling Explorations. Maci: Minimal anti-collusion infrastructure. https://maci.pse.dev/, 2023. *[pp. 35 and 36]*

[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. *[pp. 15, 49, 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. *[pp. 68 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. *[p. 30]*

[PFPB18] Louiza Papachristodoulou, Apostolos P. Fournaris, Kostas Papagiannopoulos, and Lejla Batina. Practical evaluation of protected residue number system scalar multiplication. *IACR TCHES*, 2019(1):259–282, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7341. *[p. 39]*

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019. *[pp. 28 and 48]*

[Pol78] John M Pollard. Monte carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978. *[p. 55]*

[Pre04] Dražen Prelec. A bayesian truth serum for subjective data. *Science*, 306(5695):462–466, 2004. *[p. 63]*

[PRF23] Mallesh Pai, Max Resnick, and Elijah Fox. Censorship resistance in on-chain auctions. *arXiv preprint arXiv:2301.13321*, 2023. *[p. 36]*

[Pria] Privacy and Scaling Explorations. Rate-limiting nullifier. https://rate-limiting-nullifier.github.io/rln-docs/. *[p. 35]*

[Prib] Privacy Scaling Explorations. Perpetual powers of tau. *[p. 62]*

[Pri22] Privacy & Scaling Explorations. BLS wallet: Bundling up data. https://medium.com/privacy-scaling-explorations/bls-wallet-bundling-up-data-fb5424d3bdd3, August 2022. *[p. 68]*

[PSS19] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4, 2019. *[p. 45]*

[PSW80] Carl Pomerance, John L Selfridge, and Samuel S Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980. *[pp. 49 and 58]*

[Rot21] Lior Rotem. Simple and efficient batch verification techniques for verifiable delay functions. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 382–414. Springer, Heidelberg, November 2021. *[p. 59]*

[RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996. *[pp. 8 and 38]*

[San99] Tomas Sander. Efficient accumulators without trapdoor extended abstracts. In Vijay Varadharajan and Yi Mu, editors, *ICICS 99*, volume 1726 of *LNCS*, pages 252–262. Springer, Heidelberg, November 1999. *[p. 44]*

[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. *[pp. 14, 26, 40, 50, 66, 73, and 19]*

[scr] Scroll. https://scroll.io/. *[p. 20]*

[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. *[pp. 19 and 34]*

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979. *[p. 12]*

[Sin22] Sritanshu Sinha. Can the Optimism blockchain win the battle of the rollups? *Cointelegraph*, June 2022. https://cointelegraph.com/news/can-the-optimism-blockchain-win-the-battle-of-the-rollups. *[p. 22]*

[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. *[p. 24]*

[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. *[pp. 24 and 31]*

[sta] Starknet. https://www.starknet.io/. *[p. 20]*

[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. *[pp. 13 and 7]*

[SY03] Koutarou Suzuki and Makoto Yokoo. Secure generalized Vickrey auction using homomorphic encryption. In Rebecca Wright, editor, *FC 2003*, volume 2742 of *LNCS*, pages 239–249. Springer, Heidelberg, January 2003. *[p. 35]*

[TC14] Thian Fatt Tay and Chip-Hong Chang. A new algorithm for single residue digit error correction in redundant residue number system. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1748–1751, 2014. *[p. 39]*

[TCLM21] Sri Aravinda Krishnan Thyagarajan, Guilhem Castagnos, Fabien Laguillaumie, and Giulio Malavolta. Efficient CCA timed commitments in class groups. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2663–2684. ACM Press, November 2021. *[pp. 44, 48, and 56]*

[TGB+21] Sri Aravinda Krishnan Thyagarajan, Tiantian Gong, Adithya Bhat, Aniket Kate, and Dominique Schröder. OpenSquare: Decentralized repeated modular squaring service. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 3447–3464. ACM Press, November 2021. *[p. 41]*

[Tha23] Justin Thaler. Proofs, arguments, and zero-knowledge. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf, July 2023. *[pp. 8 and 73]*

[TR19] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *arXiv preprint arXiv:1908.04756*, 2019. *[p. 21]*

[VNL+20] M.V. Valueva, N.N. Nagornov, P.A. Lyakhov, G.V. Valuev, and N.I. Chervyakov. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. *Mathematics and Computers in Simulation*, 177:232–243, 2020. *[p. 39]*

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019. *[pp. 48, 58, and 59]*

[Woo24] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, june 2024. *[p. 19]*

[Wui12] Pieter Wuille. BIP32: Hierarchical deterministic wallets. https://en.bitcoin.it/wiki/BIP_0032, February 2012. *[pp. 67 and 12]*

[XWY+21] Pengcheng Xia, Haoyu Wang, Zhou Yu, Xinyu Liu, Xiapu Luo, and Guoai Xu. Ethereum name service: the good, the bad, and the ugly. *arXiv preprint arXiv:2104.05185*, 2021. *[p. 35]*

[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. *[pp. 62, 63, 74, and 19]*

[zks] Zksync. https://www.zksync.io/. *[p. 20]*

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 $t-1 \leq d$ 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 24: 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 $(\mathtt{sid}, \mathsf{Setup}, C, (\mathsf{vk}, t, \mathsf{com}))$, delete any existing entries $(C, *, *, *, *) \in D$ and $(C, *, *) \in U$. Add $(C, \mathsf{vk}, t, \mathsf{com}, \mathtt{time} := 0)$ to D and output $(\mathtt{sid}, \mathsf{SetupResult}, C, (1))$.
- On input $(\mathtt{sid}, \mathsf{ComUpdate}, C, (\mathsf{ucom}, \pi_{\mathsf{ucom}}))$:
  1. Retrieve $(C, \mathsf{vk}, t, \mathsf{com}, \mathtt{time}) \in D$ for the maximum value of $\mathtt{time}$.
  2. If $\Pi_{\mathsf{Ref}}.\mathsf{UCVrfy}((\mathsf{crs}, \mathsf{ucom}, t), \pi_{\mathsf{ucom}}) = 1$, set $b := 1$ and add $(C, \mathsf{vk}, t, \mathsf{com} \cdot \mathsf{ucom}, \mathtt{time}{+}{+})$ to D and $(C, \mathsf{ucom}, \mathtt{time}{+}{+})$ to U. Otherwise set $b := 0$.
  3. Output $(\mathtt{sid}, \mathsf{ComUpdateResult}, C, (b))$.
- On input $(\mathtt{sid}, \mathsf{ClientInfoRecover}, P, (C))$, retrieve $(C, \mathsf{vk}, *, \mathsf{com}, \mathtt{time}) \in D$ for the maximum value of $\mathtt{time}$ and output $(\mathtt{sid}, \mathsf{ClientInfo}, P, (C, \mathsf{vk}, \mathsf{com}))$.
- On input $(\mathtt{sid}, \mathsf{UComRecover}, P, (C))$, retrieve $(C, \mathsf{ucom}, \mathtt{time}) \in U$ for the maximum value of $\mathtt{time}$ and output $(\mathtt{sid}, \mathsf{UCom}, P, (C, \mathsf{ucom}))$.

Figure 25: **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 $t \le n \le d$, $\mathcal{P} = \{P_1 \dots P_n\}$ 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 $x \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{0\}$. 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 $t \le n \le d$, 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 26: Protocol realizing $\mathcal{F}_{\mathsf{SS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model

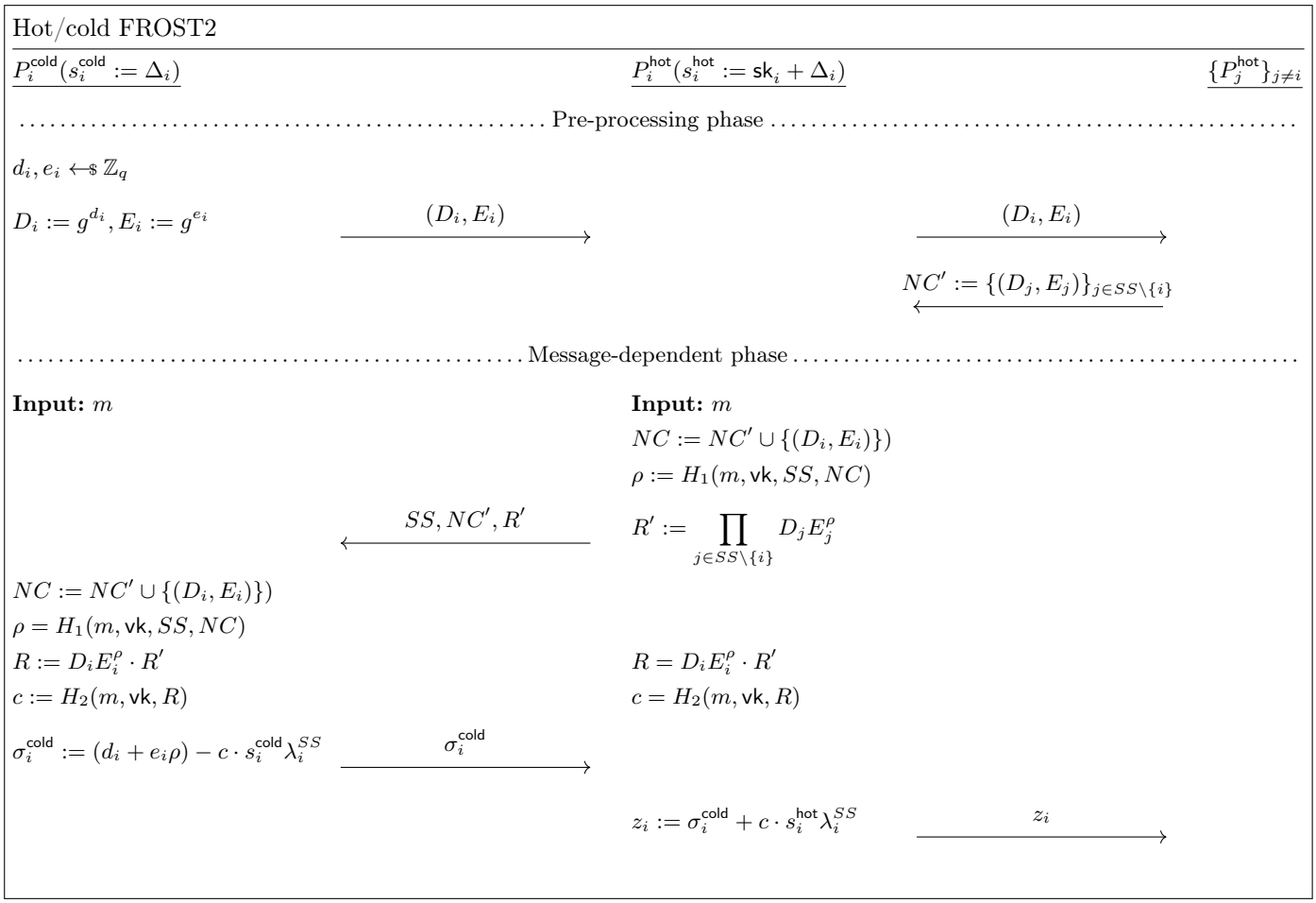

Figure 27: Our hot/cold threshold signature based on FROST2.

$\mathrm{RO}(x)$

if $\exists (x, y) \in L$
  return y
else
  $y \leftarrow\!\!\$\ \mathbb{Z}_q$
  $L = L \cup \{(x, y)\}$
  return y
fi

$\textsc{PSign}\mathrm{O}(i)$

if $i \notin HS_{\mathsf{cold}}$
  return $\perp$
fi
$d_i, e_i \leftarrow\!\!\$\ \mathbb{Z}_q$
$D_i := g^{d_i}, E_i := g^{e_i}$
$\mathsf{NMap}_i(D_i, E_i) := (d_i, e_i)$
$NC_i = NC_i \cup \{(D_i, E_i)\}$
return (D_i, E_i)

$\textsc{CSign}\mathrm{O}(i, R', lr)$

$m := lr.\mathsf{msg}$
if $lr.SS \not\subseteq [n] \vee m \notin \{0,1\}^* \vee i \notin HS_{\mathsf{cold}}$
  return $\perp$
fi
$L_{\mathsf{cold}} = L_{\mathsf{cold}} \cup \{lr\}$
$\rho \leftarrow \mathrm{RO}(1, m, \mathsf{vk}, lr.SS, lr.NC)$
$(D_j, E_j) = lr.NC(j)\ \forall j \in lr.SS$
if $i \notin CS_{\mathsf{hot}}$
  $R' = \prod_{j \in lr.SS \setminus \{i\}} D_j E_j^{\rho}$
fi
$R := D_i E_i^{\rho} \cdot R'$
$c \leftarrow \mathrm{RO}(2, m, \mathsf{vk}, R)$
$(d_i, e_i) = \mathsf{NMap}_i((D_i, E_i))$
$\sigma_i^{\mathsf{cold}} := (d_i + e_i \rho) - c \cdot s_i^{\mathsf{cold}} \lambda_i^{SS}$
if $\sigma_i^{\mathsf{cold}} \neq \perp$
  $S_{\mathsf{cold}}(lr) = S_{\mathsf{cold}}(lr) \cup \{i\}$
fi
return σ_i^{cold}

$\textsc{HSign}\mathrm{O}(i, \sigma_i^{\mathsf{cold}}, lr)$

$m := lr.\mathsf{msg}$
if $lr.SS \not\subseteq [n] \vee m \notin \{0,1\}^* \vee i \notin HS_{\mathsf{hot}}$
  return $\perp$
fi
$L_{\mathsf{hot}} = L_{\mathsf{hot}} \cup \{lr\}$
$\rho \leftarrow \mathrm{RO}(1, m, \mathsf{vk}, lr.SS, lr.NC)$
$(D_j, E_j) = lr.NC(j)\ \forall j \in lr.SS$
$R := \prod_{j \in lr.SS} D_j E_j^{\rho}$
$c \leftarrow \mathrm{RO}(2, m, \mathsf{vk}, R)$
if $i \notin CS_{\mathsf{cold}}$
  $(d_i, e_i) = \mathsf{NMap}_i((D_i, E_i))$
  $\sigma_i^{\mathsf{cold}} = (d_i + e_i \rho) - c \cdot s_i^{\mathsf{cold}} \lambda_i^{SS}$
fi
$z_i := \sigma_i^{\mathsf{cold}} + c \cdot s_i^{\mathsf{hot}} \lambda_i^{SS}$
$\sigma_i^{\mathsf{hot}} := (R, z_i)$
if $\sigma_i^{\mathsf{hot}} \neq \perp$
  $S_{\mathsf{hot}}(lr) = S_{\mathsf{hot}}(lr) \cup \{i\}$
fi
return σ_i^{hot}

Figure 28: Oracles for HC-SUF-2 experiment (see **??**) tailored to the HC-FROST2 protocol (Figure 27).