ABSTRACT

Title of Dissertation:	PRACTICAL CRYPTOGRAPHY FOR BLOCKCHAINS: SECURE PROTOCOLS WITH MINIMAL TRUST
	Noemi Sara Maria Glaeser Doctor of Philosophy, 2024
Dissertation Directed by:	Professor Jonathan Katz Department of Computer Science

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 cover two on-chain applications: First, I show how to improve the security of a class of coin mixing protocols by giving a formal security treatment of the construction paradigm and patching the security of an existing, insecure protocol. Second, I show how to construct practical on-chain protocols for a large class of elections and auctions which simultaneously offer fairness and non-interactivity without relying on a trusted third party. Finally, I look to the edges of the blockchain and formalize new design requirements for the problem of backing

up high-value but rarely-used secret keys, such as those used to secure the reserves of a cryptocurrency exchange, and develop a protocol which efficiently meets these new challenges.

All of these works will be deployed in practice or have seen interest from practitioners. These examples show that advanced cryptography has the potential to meaningfully nudge the blockchain ecosystem towards increased security and reduced trust.

PRACTICAL CRYPTOGRAPHY FOR BLOCKCHAINS: SECURE PROTOCOLS WITH MINIMAL TRUST

by

Noemi Sara Maria Glaeser

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:
Professor Jonathan Katz, Chair/Advisor
Dr. Giulio Malavolta, Co-Advisor
Professor Dana Dachman-Soled, Dean's Representative
Professor Ian Miers
Professor Gabriel Kaptchuk

Acknowledgments

It takes a village to raise a child, and apparently to raise a Ph.D. as well. My graduate career has been marked by uncountable people who have given me the energy, support, and confidence to keep going in the graduate program over the last five-and-a-half years. I hope they know who they are, and I will try to list as many as I can here.

First and foremost, I want to thank my advisors for taking me on as a student. To Jonathan Katz, who took me on with only limited experience in cryptography; and Giulio Malavolta, who agreed to co-advise me as a relative unknown. Even though I was constantly coming and going, Jon and Giulio always answered when I reached out with questions and supported me in applying to internships, traveling to conferences, etc. Thanks also to Gilles Barthe, my first advisor at MPI-SP, who was patient with me when I was still trying to figure out what sort of security and privacy research I was interested in, helped put me in Giulio's hands, and continued to be willing to support me even when I was no longer his student. And thank you to Duncan Buell, who, back in 2015, convinced me to study math and computer science.

I am grateful to the National Science Foundation for the Graduate Research Fellowship which helped fund me during most of graduate school, and to the Maryland-Max Planck program for allowing me to travel widely, meet people from all over the world, and collaborate across many institutions. I am also indebted to NTT Research and a16z crypto for amazing internship experiences which had a great effect on shaping my research agenda and led to a large portion of the research in this dissertation.

The wider faculty at UMD, MPI-SWS, MPI-SP, and Bocconi University also left a mark as role models, lunch and coffee companions, and by dispensing occasional doses of wisdom and experience. The same goes for the researchers and visitors at NTT Research and a16z crypto who I had the privilege of coming into contact with during my summer internships. Thank you especially to all of my coauthors, each of whom taught me something new about how to do and present great research. Finally, I am grateful to the other students and young researchers I met at these institutions or through these internships, research visits, and conferences for creating a kind, fun, and supportive research community. In particular, I want to thank Erica Blum, Miranda Christ, Jason Fan, Phillip Gajland (and the rest of the kicker squad at MPI-SP), Doruk Gür, Lillian Huang, Hannah Keller, Hunter Kippen, Lisa Masserova, Vedant

Nanda, Michael Rosenberg, István Seres, Noel Warford, and the Bocconi Clowns (Damiano Abram, Pedro Branco, Ruta Jawale, Darya Kaviani, Tamer Mour, and Nikolaj Schwartzbach).

On the personal side, thank you to my family, especially my parents for being my biggest fans and always encouraging and supporting me in my endeavors, and my sister Tullia for her pragmatism and reminding me when things don't merit my constant worrying. Thanks also to my many flatmates over the last five and a half years who always made my various houses (some seven or eight of them) feel like homes; and especially Sarah, who reminds me to eat and sleep even when I feel very overwhelmed. Thank you to my friends (especially Joelle, Liz, and Noémie; Marie and Marisa; Makana; and Spyral), who remind me I'm a whole person and not just a grad student.

On a technical level, thank you to Alex Block and Doruk Gür for feedback on naysayer proofs for FRI (Section 3.4.2) and Dilithium (Section 3.4.3), respectively, and to Joe Bonneau, Jonathan Katz, Giulio Malavolta, and István Seres for proofreading and comments on larger sections of this dissertation.

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. Under submission.

[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.

[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. Also under submission.

[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.

Table of Contents

Chapter 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 Chapter 2, I introduce the necessary background and building blocks used throughout this dissertation.

Chapter 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 Chapters 4 and 5, I move to on-chain applications. Chapter 4 analyzes the security of a class of coin mixing services which require minimal functionalities of the underlying blockchain, rendering them highly interoperable. I discuss two gaps in the formal treatment of a previous protocol and attacks which exploit them. To close this gap, I introduce a new primitive called *blind conditional* *signatures* (BCS) and use it to prove the security of two new coin mixing protocols. Chapter 5 describes 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 present in previous 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 reached out to Optimism, one of the largest L2s in the Ethereum ecosystem, to discuss the use of Cicada for their retroactive public goods funding (retroPGF) vote.[1]

Finally, Chapter 6 moves off-chain and looks at securing the edges of the system: cryptocurrency wallets. I envision a system that allows users to conveniently back up their rarely-used, high-value keys (such as the signing keys of high-balance wallets). This new setting necessitates a novel set of design requirements. I develop new security definitions that capture them and construct a UC-secure protocol that implements threshold BLS signatures in our new model. The protocol is practically efficient for the envisioned large numbers of custodians: for a 67-out-of-100 threshold configuration, setup takes 170ms and signing less than 1ms. This design was created with collaborators from Lit Protocol[2] and the Linux Foundation and is in the process of being adopted in production by the former. An Apache 2.0-licensed implementation is also publicly available in Hyperledger Labs[3], a popular open source organization.

I conclude in Chapter 7 by discussing potential future directions for these three works and a broader outlook on the role of cryptography in blockchains.

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

Chapter 2

Preliminaries

Contents

2.1 Notation

I use $[n]$ to denote the range $\{1, \dots, n\}$. 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. I use the uppercase variable X for the free variable of a polynomial, e.g., $f(X)$. I use a calligraphic font, e.g., $\mathcal{S}$ or $\mathcal{X}$, to denote sets or domains. When applying an operation to two sets of equal size ℓ I mean pairwise application, e.g., $\mathcal{Z} = \mathcal{X} + \mathcal{Y}$ means $z_i = x_i + y_i \ \forall i \in [\ell]$. Sampling an element x uniformly at random from a set $\mathcal{X}$ is written as $x \leftarrow\!\!\$ \ \mathcal{X}$. I 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 I want to be explicit about the randomness r used, I write $y \leftarrow \mathsf{Alg}(x; r)$. An algorithm is PPT if it runs in probabilistic polynomial time and a function is *negligible* if it vanishes faster than any polynomial. I 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 a negligible function $\mathsf{negl}(\lambda)$.

For a non-interactive proof system Π, I 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 $\Pi.\mathsf{Verify}(x, \pi)$ and outputs a bit b.

I 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).

2.2 Non-interactive Proof Systems

A *language* $\mathcal{L} \subseteq \{0,1\}^*$ is a set of elements. In this dissertation, I will only consider the set of NP languages. Every language $\mathcal{L} \in \mathsf{NP}$ has a corresponding polynomially-decidable relation $\mathcal{R}_\mathcal{L}$ (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. $(x, w) \in \mathcal{R}_\mathcal{L}$, with $w \in \mathsf{poly}(|x|)$. Conversely, I may refer to the language corresponding to a relation $\mathcal{R}$ as $\mathcal{L}_\mathcal{R}$.

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

- $\underline{\mathsf{Setup}(1^\lambda) \to \mathsf{crs}}$*: Given a security parameter, output a common reference string* crs*. This algorithm might use private randomness (a trusted setup).*
- $\underline{\mathsf{Prove}(\mathsf{crs}, x, w) \to \pi}$*: Given the* crs*, an instance* x*, and witness* w *such that* $(x, w) \in \mathcal{R}_\mathcal{L}$*, output a proof* π*.*
- $\underline{\mathsf{Verify}(\mathsf{crs}, x, \pi) \to \{0,1\}}$*: Given the* crs *and a proof* π *for the instance* x*, output a bit indicating accept or reject.*

(Perfect) completeness *requires that for all* $(x, w) \in \mathcal{R}_\mathcal{L}$*,*

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

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

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

I refer the reader to [Tha23b, 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, I require succinct proof systems only to have proofs which are sublinear in the size of the witness and "work-saving" verification [Tha23a]:

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

2.3 Bilinear Pairings

Definition 4 (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*. Let* $g_1, h_1 \in \mathbb{G}_1$ *and* $g_2, h_2 \in \mathbb{G}_2$ *be generators of their respective groups. Bilinearity requires the map* e *to have the following properties:*

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

(Note this implies $e(g_1^a, g_2) = e(g_1, g_2^a) = e(g_1, g_2)^a$ *for all* $a \in \mathbb{Z}_p^*$*.)*

Pairings used in cryptography are generally also required to be *non-degenerate*, i.e., $e(g_1, g_2) \neq 1$.

Based on the (in)equality of the groups $\mathbb{G}_1, \mathbb{G}_2$ and the (non-)existence of an efficiently computable homomorphism between them, pairings can be divided into three types. The constructions in this dissertation will use a type-3 pairing, which is asymmetric ($\mathbb{G}_1 \neq \mathbb{G}_2$) and has 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.*

2.4 Shamir Secret Sharing

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

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

- $\underline{\{s_1, \ldots, s_n\} \leftarrow \mathsf{Share}(s, t, n)}$*: Given a secret* $s \in \mathbb{Z}_p$ *and* $t \leq n \in \mathbb{N}$*, compute a* t*-out-of-*n *sharing of* s *by choosing a random degree-*$(t-1)$ *polynomial* $f(X) \in \mathbb{Z}_p[X]$ *such that* $f(0) = s$*. For* $i \in [n]$*, compute* $s_i := (i, f(i))$*.*
- $\underline{\{s', \perp\} \leftarrow \mathsf{Reconstruct}(S, t, n)}$*: 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:

- $\{s_k, \bot\} \leftarrow \mathsf{Interpolate}(S, k, t, n)$: If $|S| < t$, output $\bot$. 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 $s_k := (k, f(k))$.

2.5 Digital Signatures

A digital signature scheme is a triple of algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$. The key generation algorithm $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$ outputs a verification-signing key pair. The owner of the signing key sk can compute signatures on a message m by running $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$, which can be publicly verified using the corresponding verification key vk by running $\mathsf{Verify}(\mathsf{vk}, m, \sigma)$.

2.5.1 BLS Signatures

A popular digital signature scheme is the BLS signature scheme, which uses bilinear pairings (Section 2.3).

Construction 2 (BLS signature scheme [BLS01])**.**

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda)$: *Given the security parameter* 1^λ*, generate elliptic curve groups* $\mathbb{G}_1, \mathbb{G}_2$ *of prime order* p *(where* $\log p = \lambda$*) with generators* g_1 *and* g_2*, respectively, and an efficiently computable*[1] *asymmetric pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$*. Sample* $x \leftarrow\!\$\ \mathbb{Z}_p$ *and output the keypair consisting of signing key* $\mathsf{sk} := x$ *and verification key* $\mathsf{vk} := g_2^x$*.*

  For simplicity, we add the group description $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e)$ *to the verification key* vk*. Let* $H : \{0,1\}^* \to \mathbb{G}_1$ *be a public hash function.*

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

- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{vk}, m, \sigma)$: *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]).

[1] i.e., in time $\mathsf{poly}(\lambda)$

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

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

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

- $\underline{\sigma \leftarrow \mathsf{Reconstruct}(S := \{(i, \sigma_i)\})}$: Let $S' \subseteq S$ be the set of valid partial signatures in S. If $|S'| < t$, output $\perp$. 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.

2.6 KZG Polynomial Commitments

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

Construction 3 (KZG polynomial commitments [KZG10])**.**

- $\underline{\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda, d)}$: *Given a security parameter 1^λ, generate elliptic curve groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p (where $\log p = \lambda$) with generators g_1 and g_2, respectively, and an efficiently computable asymmetric pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$. To allow commitments to polynomials in $\mathbb{Z}_p[X]$ with degree at most d, sample $\tau \leftarrow\!\!\$\ \mathbb{Z}_p$ and output* $\mathsf{crs} := \{g_1, g_1^{\tau}, g_1^{\tau^2}, \ldots, g_1^{\tau^d}, g_2, g_2^{\tau}\}$.

  For simplicity, we add the group description $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e)$ to crs.

- $\underline{\mathsf{com}_f \leftarrow \mathsf{Com}(\mathsf{crs}, f(X))}$: *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* $q_i(X) := \frac{f(X)-f(i)}{X-i} \in \mathbb{Z}_p[X]^2$. *Then compute* $\mathsf{com}_{q_i} \leftarrow \mathsf{Com}(\mathsf{crs}, q_i(X))$ *and output* $(f(i), \mathsf{com}_{q_i}) \in \mathbb{Z}_p \times \mathbb{G}_1$.

- $\underline{\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{crs}, \mathsf{com}_f, i, y, \pi_i):}$ *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 the symmetric case when $\mathbb{G}_1 = \mathbb{G}_2$, which in turn implies DDH in $\mathbb{G}_1$.

2.7 Pedersen Commitments

Next, I recall Pedersen commitments [Ped92], a commitment scheme which is unconditionally (information-theroetically) hiding and computationally binding (by the discrete logarithm assumption on $\mathbb{G}$).

Construction 4 (Pedersen commitment scheme [Ped92])**.** *Let* $\mathbb{G}$ *be a group of order* p *and* g, h *be generators of* $\mathbb{G}$. *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* $\mathsf{com} := g^x h^r$ *and* *decommitment information* (x, r).

- $\underline{(x, r) \leftarrow \mathsf{Open}(\mathsf{com}, \mathsf{decom}):}$ *To open* com, *directly output* $\mathsf{decom} = (x, r)$.

- $\underline{\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{com}, x, r):}$ *To confirm the opening of* com *to* x, *it suffices to check that* $\mathsf{com} = g^x h^r$.

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]. I refer to this protocol as Π_{ped} and present it in Figure 2.1.

2.8 Universal Composability (UC) Framework

In the universal composability (UC) framework [Can20], 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

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

PoK of Pedersen opening (Π_{ped})

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

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

1. The prover samples $s_1, s_2 \leftarrow\$ \mathbb{Z}_p$ 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 $\pi_{\mathsf{ped}} := (a, c, (t_1, t_2))$.

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

Figure 2.1: The proof system Π_{ped} used to prove knowledge of the opening to a Pedersen commitment [Oka93].

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 [CLOS02a]. 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 $\textsc{real}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)$ the random variable describing the output of the real-world execution of Π with security parameter λ and input x in the presence of adversary $\mathcal{A}$ and environment $\mathcal{E}$. We write the corresponding distribution ensemble as $\{\textsc{real}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$. The output of the ideal-world interaction with ideal functionality $\mathcal{F}$, adversary (simulator) $\mathcal{S}$, and environment $\mathcal{E}$ is represented by the random variable $\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)$ and corresponding distribution ensemble $\{\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$.

The actions each party can take are summarized below:

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

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

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

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

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

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

Definition 5. *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* $\{\textsc{real}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$ *and* $\{\textsc{ideal}_{\mathcal{F},\mathcal{S},\mathcal{E}}(\lambda, x)\}_{\lambda\in\mathbb{N},x\in\{0,1\}^*}$ *are computationally indistinguishable.*

Chapter 3

Naysayer Proofs*

Contents

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 short proof[1] that the claimed result is correct. This computation can even depend on secret inputs not known to the verifier by relying on zero-knowledge proofs (i.e., zkSNARKs).

[1] More precisely, a succinct non-interactive argument of knowledge, or SNARK.

* *Portions of this section have been adapted from [SGB24].*

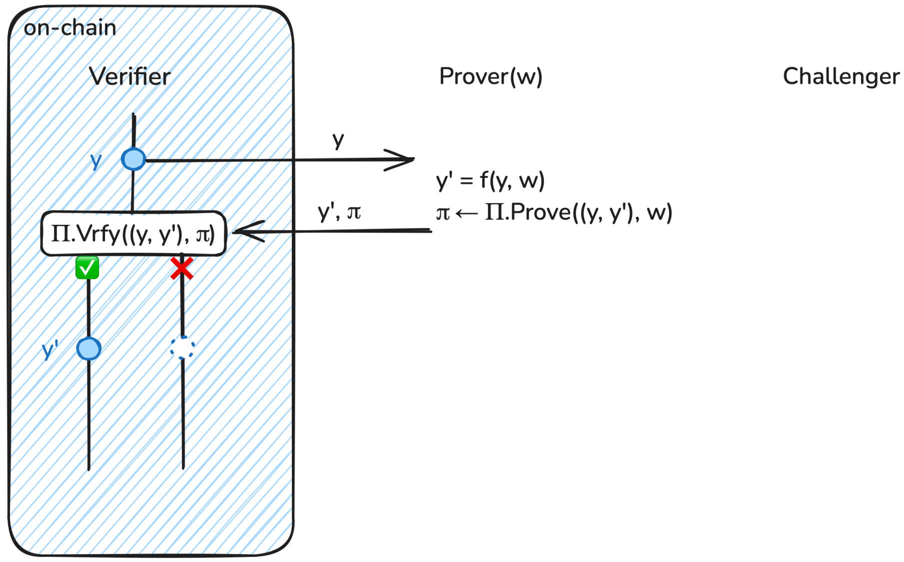

Figure 3.1: **Using VC to move computation off-chain.** The off-chain "prover" applies the function f to input y and potentially an auxiliary off-chain input w to get the result $y' = f(y, w)$. (In the case of a zk-rollup, f is the state transition function, y is the previous on-chain state, w is a batch 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.

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 3.1). A motivating application are so-called "zk"-*rollups*[2] [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.[3]

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.[4]

This state of affairs motivates us to propose a new paradigm called *naysayer proofs* (Figure 3.2). In this paradigm, the verifier (e.g., a rollup smart

[2]The "zk" part of the name is often a misnomer, since these services do not necessarily offer the zero-knowledge property (and in fact most do not). Instead, the term is used by

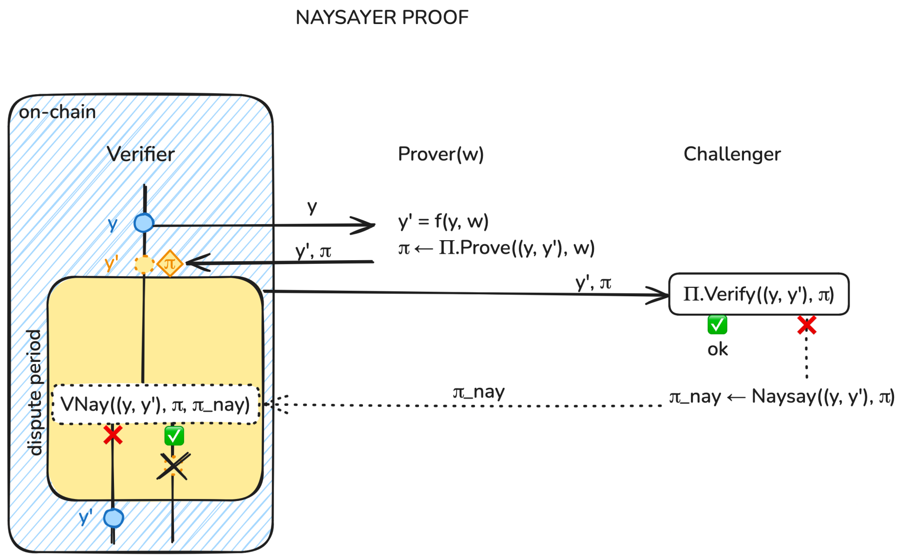

Figure 3.2: **The naysayer proof approach.** As in VC, the off-chain prover computes $y' = f(y, w)$ and π, which it posts on-chain. This time, the proof is *not* verified on-chain, but is provisionally accepted while waiting for the challenge period to pass. Any party can verify π off-chain and, if it 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.

contract) optimistically accepts a submitted proof without verifying its correctness. Instead, any observer can check the proof off-chain and, if needed, prove its *incorrectness* to the verifier by submitting a *naysayer proof.* The verifier then checks the naysayer proof and, if it is correct, rejects the original proof. Otherwise, if no party 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 (just like the related fraud proof paradigm; 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

practitioners to emphasize succinctness, which is the more relevant property in practice.

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

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

verifier by reducing the cost of the verification procedure in fraudulent cases. At worst, checking the naysayer proof is equivalent to verifying the original proof (this is the trivial naysayer construction).

Naysayer proofs enable other interesting trade-offs. For instance, naysayer proofs can be instantiated with a lower security level than the original proof system. This is because a violation of the naysayer proof system's soundness undermines only the *completeness* of the original proof system. For an application like a rollup service, this would result 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. Before that, 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$^+$18, AAB$^+$24] 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 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 3.3). (The term "fraud proof" or "fault proof" refers to these messages.[5]) 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 3.4), 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 committed on the layer-1 chain.

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

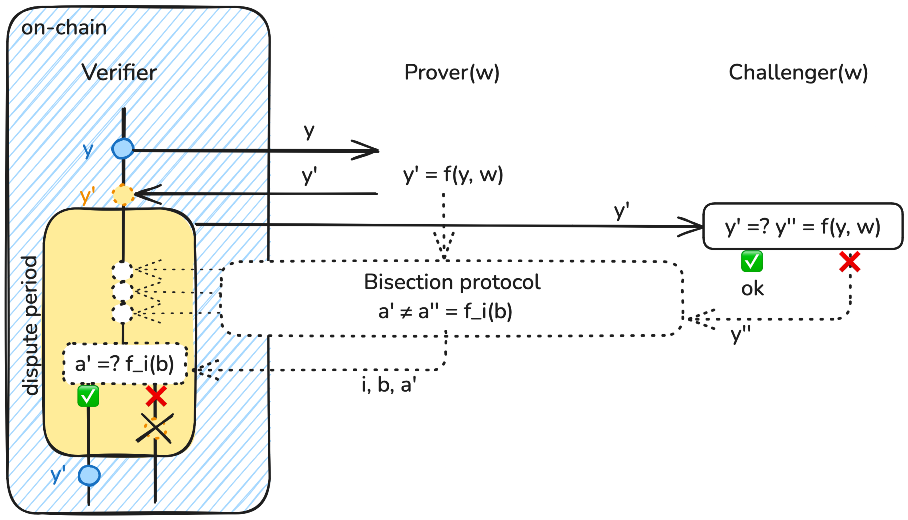

Figure 3.3: **Interactive fraud "proofs".** Like naysayer proofs, fraud "proofs" make use of challengers and challenge periods. Again, the off-chain "prover" computes $y' = f(y, w)$, but without providing any proof of correctness π. During the challenge period, anyone (with access to w, e.g., the batch 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.

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 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. A prover who performs the computation honestly has no incentive to attach an incorrect proof[6], since that would mean it wasted computational power to compute the result, but would forfeit the reward (and likely incur some additional penalty).

We compare classic verifiable computation, fraud proofs, and our naysayer proofs in Table 3.1. We discuss the main differences in more detail below.

Assumptions. Both fraud proofs and naysayer proofs work under an optimistic

[6]It is possible that an honest prover will still attach an incorrect proof if, for example, the proof generation software has a bug.

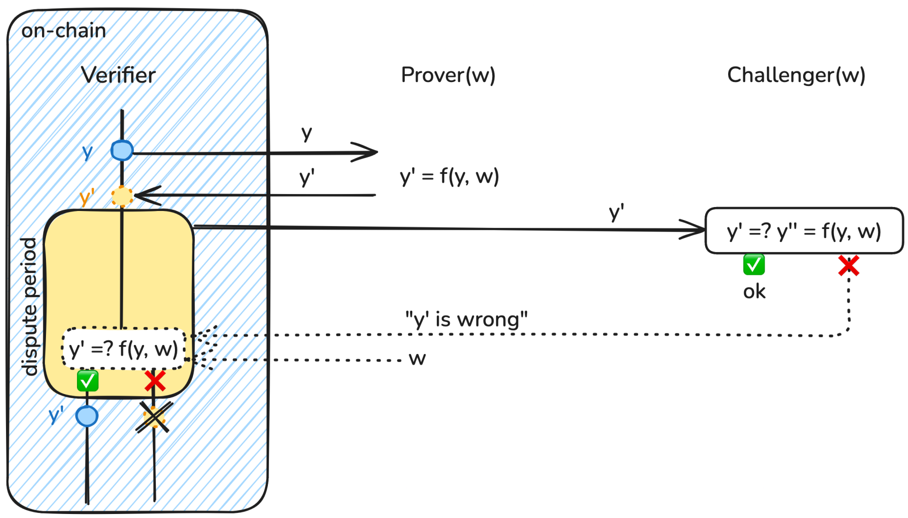

Figure 3.4: **Non-interactive fraud "proofs".** The non-interactive version functions similarly, except that there is no bisection game. Instead, the on-chain verifier simply re-executes the entire computation $f(y, w)$ on-chain to decide whether or not to reject y'. Again, if the challenge period elapses without any challenges, y' is accepted.

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.

On-chain interaction. 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-chain) verifier. VC offers the strongest form of non-interactivity, since it consists of one round *total* (for the original computation and (non-existent) challenge phase). At the other end of the spectrum, 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 dispute resolution 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

	VC	fraud proof (interactive)	fraud proof (non-interactive)	naysayer proof
No optimistic assumption	●	○	○	○
Non-interactive	●	○	◐	◐
Off-chain f	●	●	◐	●
Off-chain $\Pi.\mathsf{Verify}$	○	-	-	●
Witness-independent challenge	-	○	○	●
Witness-independent resolution	●	◐	○	●
No $\Pi.\mathsf{Prove}$	○	●	●	○

Table 3.1: Trade-offs between VC, fraud proofs, and naysayer proofs.

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.

On-chain computation & witnesses. 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.

Underlying proof system. 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 $\mathsf{Verify}(\mathsf{crs}, (x, y), \pi) = 0$, 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 6 (Naysayer proof)**.** *Given a non-interactive proof system* $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *for an NP language* $\mathcal{L}$*, the naysayer proof system corresponding to* Π *is a tuple of PPT algorithms* $\Pi_{\mathsf{nay}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Naysay}, \mathsf{VerifyNay})$ *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* $(x, w) \in \mathcal{R}_{\mathcal{L}}$*, 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 $\exists w$ s.t. $(x, w) \in \mathcal{R}_{\mathcal{L}}$ using the proof system Π, output a naysayer proof π_{nay} disputing π. This algorithm may also make use of some (private) trapdoor information $\mathsf{td}_{\mathsf{nay}} \subseteq w$.*

$\mathsf{VerifyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \pi_{\mathsf{nay}}) \to \{0, \perp\}$**:** *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 ($\perp$).*

A trivial naysayer proof system always exists in which $\pi_{\mathsf{nay}} = \top$, $\pi' = (\pi, \perp)$, and VerifyNay simply runs the original verification procedure, outputting 0 if $\Pi.\mathsf{Verify}(\mathsf{crs}, x, \pi) = 0$ and $\perp$ otherwise. We say a proof system Π is *efficiently naysayable* if there exists a corresponding naysayer proof system Π_{nay} such that VerifyNay is asymptotically faster than Verify. If VerifyNay is only concretely faster than Verify, 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 $\mathsf{td}_{\mathsf{nay}} = \perp$, we say Π_{nay} is a *public* naysayer proof (see Section 3.4.4 for an example of a non-public naysayer proof).

Definition 7 (Naysayer completeness)**.** *Given a proof system Π, a naysayer proof system $\Pi_{\mathsf{nay}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Naysay}, \mathsf{VerifyNay})$ is* complete *if, for all honestly generated $\mathsf{crs}, \mathsf{crs}_{\mathsf{nay}}$ and all values of aux,[7] 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, the following expression equals 1:*

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

Definition 8 (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 $\mathsf{td}_{\mathsf{nay}}$, the following expression is bounded by $\mathsf{negl}(\lambda_{\mathsf{nay}})$:*[8]

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

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

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

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).

Lemma 1. *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 $\mathsf{poly}(|x|, |w|)$ verification complexity has a succinct naysayer proof.*

Proof. Given any proof system Π, the evaluation of $\Pi.\mathsf{Verify}(\mathsf{crs}, \cdot, \cdot)$ 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 $\Pi.\mathsf{Verify}(\mathsf{crs}, x, \pi)$. Output $\pi' = (\pi, \mathsf{aux})$.

$\mathsf{Naysay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \mathsf{td}_{\mathsf{nay}})$**:** Parse $\pi' = (\pi, \mathsf{aux})$[9] and output $\pi_{\mathsf{nay}} := \top$ if $\mathsf{aux} = \mathsf{aux}' \| 0$. Otherwise, evaluate $\Pi.\mathsf{Verify}(\mathsf{crs}, x, \pi)$. If the result is not 1, search aux to find an incorrect wire assignment for some gate $G_i \in C$. Output $\pi_{\mathsf{nay}} := i$.

$\mathsf{VerifyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), \pi_{\mathsf{nay}})$**:** 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 1. If $\Pi.\mathsf{Verify}(\mathsf{crs}, x, \pi) \neq 1$, then we have two cases: If aux is consistent with a correct evaluation of $\Pi.\mathsf{Verify}(\mathsf{crs}, x, \pi)$, either $\mathsf{aux} = \mathsf{aux}' \| 0$ (and $\mathsf{VerifyNay}$ 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 $\Pi.\mathsf{Verify}(\mathsf{crs}, x, \pi) = 1$. Since

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

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 $\mathsf{VerifyNay}(\mathsf{crs}_{\mathsf{nay}}, (x, \pi'), i)$ to output 0.

Succinctness of π_{nay} follows from the fact that $|i| = \log|\Pi.\mathsf{Verify}(\mathsf{crs}, \cdot, \cdot)| = \mathcal{O}(\log(|x|, |w|)) \in o(|x| + |w|)$ and that the runtime of VerifyNay 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 succinct proof systems, the generic construction even allows efficient (sublinear) naysaying, since the runtime of Naysay depends only on the runtime of Π.Verify, which is sublinear if Π is succinct.

Notice that although the syntax gives $\pi' = (\pi, \mathsf{aux})$ as an input to the VerifyNay algorithm, in the generic construction the algorithm does not make use of π. Thus, if the naysayer rollup from Figure 3.2 were instantiated with this generic construction, π would not need to be posted on-chain since the on-chain verifier (running the VerifyNay 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. In Section 3.5, we will discuss how to leverage this insight to reduce the storage costs of a naysayer rollup.

3.4 Constructions

Our first construction (Section 3.4.1) is a concrete example of the generic naysayer construction from Theorem 1, applied to Merkle trees. We then highlight three naysayer proof constructions which take advantage of repetition in the verification procedure to achieve better naysayer performance: the FRI polynomial commitment scheme (Section 3.4.2) and two post-quantum signature schemes (Section 3.4.3).

Our first construction (Section 3.4.1) is a concrete example of the generic naysayer construction from Theorem 1, applied to Merkle trees. We then highlight three naysayer proof constructions which take advantage of repetition in the verification procedure to achieve better naysayer performance: the FRI polynomial commitment scheme (Section 3.4.2) and two post-quantum signature schemes (Section 3.4.3).

Many proof systems have some repetitive structure in their verification algorithm. This structure allows for more efficient naysaying. A common example is a verification check which is a conjunction of multiple independent checks: since all the statements in the conjunction must hold for a proof to be accepted, for naysaying it suffices to point out a single clause of the conjunction which does not hold. Our constructions in this section fall into this category. Other examples are Plonk proofs [GWC19], whose verification requires multiple bilinear pairing checks, or proofs with multi-round soundness amplification.

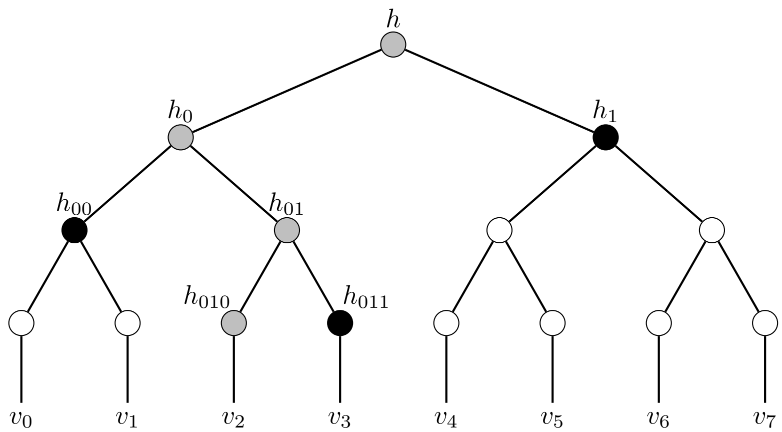

Figure 3.5: Each node in a Merkle tree consists of a hash of its children. The root h is a commitment to the vector of leaves $(v_0, v_1, \ldots, v_7)$. An opening proof for the element v_2 is its copath (black nodes); the "verification trace" for the proof is the path (gray nodes).

Our first construction (Section 3.4.1) is a concrete example of the generic naysayer construction from Theorem 1, applied to Merkle trees. 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). Finally, 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.

3.4.1 Merkle Commitments

	Proof size	**Verification**
Original	$\log n\ \mathbb{H}$	$\log n\ \mathbb{H}$
Naysayer	$\log\log n\ \mathbb{B}$	$1\mathbb{H}$

Table 3.2: Cost savings of the naysayer paradigm applied to Merkle proofs. $\mathbb{H} =$ hash output size/hash operations, $\mathbb{B} =$ bits.

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 to a vector $\mathbf{v}$ of elements as shown in Figure 3.5, 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 gray 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

	Proof size	**Verification**
Original	$\mathcal{O}\left(\lambda \log^2 d\right) \mathbb{H} + \mathcal{O}\left(\lambda \log d\right) \mathbb{F}$	$\mathcal{O}\left(\lambda \log^2 d\right) \mathbb{H} + \mathcal{O}\left(\lambda \log d\right) \mathbb{F}$
Naysayer	$2 \log(q \log d) + 1\ \mathbb{B}$	best: $\mathcal{O}\left(1\right) \mathbb{F}$ worst: $\mathcal{O}\left(\log d\right) \mathbb{H}$

Table 3.3: Cost savings of the naysayer paradigm applied to FRI opening proofs. $\mathbb{H}$ = hash output size/hash operations, $\mathbb{F}$ = field element size/operations, $\mathbb{B}$ = bits.

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 $\pi_{\mathsf{nay}}^{\mathsf{FRI}} = (i, j, k)$ 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[10] or a few arithmetic operations over $\mathbb{F}$ in the latter.

This approach can lead to incredible concrete savings: According to [Hab22], for $\lambda = 128$, $d = 2^{12}$,[11] $\rho = 2^{-3}$, $q = 91$, $\delta = 9$, the size of a vanilla FRI opening proof (i.e., without concrete optimizations) can be estimated at around 322KB. A naysayer proof for the same parameter settings is $2 \log(q \log d)+1 \approx 2 \cdot 10+1 = 21$ bits < 3 bytes.

3.4.3 Post-quantum Signature Schemes

	Proof size	**Verification**
Original	$\mathcal{O}(\lambda)\,\mathbb{F}$	$\mathcal{O}(\lambda)\,\mathbb{F} + 1\mathbb{H}$
Naysayer	$2 + \log k + \log d\ \mathbb{B}$	best: $\mathcal{O}(1)\mathbb{F}$ worst: $\mathcal{O}(\lambda)\,\mathbb{F} + 1\mathbb{H}$

Table 3.4: Cost savings of the naysayer paradigm applied to CRYSTALS-Dilithium signatures. $\mathbb{H}$ = hash output size/hash operations, $\mathbb{F}$ = field element size/operations, $\mathbb{B}$ = bits. Since the parameter k depends on λ and d is a constant, $|\pi_{\mathsf{nay}}| \in \mathcal{O}(\log \lambda)$.

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.[12] 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

[10] 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.

[11] This is smaller than most polynomial degrees used in production systems today.

[12] 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).

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:

$$\|\mathbf{z}_1\|_\infty < \beta \wedge \|\mathbf{z}_2\|_\infty < \beta \wedge c = H(M, \mathbf{w}, \mathsf{pk}). \tag{3.1}$$

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 $d = 256$.) 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 (3.1) is a conjunction, the naysayer prover must show that

$$(\exists z_i \in \mathbf{z}_1, \mathbf{z}_2 : \|z_i\|_\infty > \beta) \vee c \neq H(M, \mathbf{w}, \mathsf{pk}). \tag{3.2}$$

If the first check of Equation (3.1) 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 $z_i = \cdots + \alpha_j X^j + \cdots \in \mathbf{z}_b$.)[13]

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 (3.2), 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

[13] 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 $\mathbf{z}_1, \mathbf{z}_2$.

	Proof size	**Verification**
Original	$\mathcal{O}(\sqrt{n})\,\mathbb{G}$	$\mathcal{O}(n)\,\mathbb{G}$
Naysayer	$\log n\ \mathbb{B} + 3\mathbb{G} + 1\mathbb{F}$	$\mathcal{O}(1)\,\mathbb{G} + 1\mathbb{H}$

Table 3.5: Cost savings of the naysayer paradigm applied to Bayer-Groth shuffles. $\mathbb{H}$ = hash output size/hash operations, $\mathbb{G}$ = group element size/operations, $\mathbb{B}$ = bits.

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 $f : [n] \to [n]$.

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[14]. 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 $\mathcal{O}(\log n)$-sized naysayer proof with $\mathcal{O}(1)$-verification, yielding in exponential savings compared to verifying the original Bayer-Groth shuffle proof.

3.4.5 Summary

We showed the asymptotic cost savings of the verifiers in the four examples discussed in Sections 3.4.1 to 3.4.4 in their respective tables. Note that the verifier speedup is exponential for verifiable shuffles and logarithmic for the Merkle and 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. (Note this calculation does not include the

[14] 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.

size of aux, but we will see in the next section that aux does not meaningfully impact the proof size for the verifier.) Furthermore, in most cases, the naysayer proof consists of an *integer* index or indices rather than group or field elements. Representing the former requires only a few bits compared to the latter (which are normally at least λ bits long), so in practice, naysayer proofs can offer *practically* smaller proofs sizes even when they are not asymptotically smaller. This can lead to savings even when the original proof is constant-size (e.g., a few group elements).

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 $\mathsf{VerifyNay}$ instead of Verify. 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

In the months since the publication of the original paper [SGB24], naysayer proofs have already received attention from various groups of practitioners hoping to deploy them in production. We see many interesting directions for investigation, both for immediate deployments and to improve our understanding of this new paradigm 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. For example, one can imagine a different fraud/naysayer proof hybrid in which the prover sends only the instance x, but a challenger can provide a corrected instance x' and proof for the new statement (i.e., "nesting" a zk-rollup inside an optimistic rollup). This can reduce both prover and verifier costs, but makes issuing a challenge less accessible than in either VC or fraud proofs.

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). A zkVM expresses each language as program P in a standard instruction set and, given an instance (a, b) s.t. $b = P(a)$, proves a conjunction of correct state transitions based on P and the instruction set. Given this repetitive structure of the proven relation, challenging an incorrect instance (a, b) s.t. $b \neq P(a)$ can be done extremely efficiently. Designing proof systems to efficiently disprove state transitions of popular zkVM instruction sets is an interesting direction for future work.

Other naysayer constructions. In all of our application-specific naysayer constructions, the verification of the base proof systems was a conjunction. An interesting question to explore is whether there are other application-specific naysayer proofs which are particularly efficient or useful.

Computing aux. A drawback of (generic) naysayer proofs is that the prover must additionally run $\Pi.\mathsf{Verify}$ to create the verification trace aux. It would be very useful to separate or outsource this computation to another party to reduce the added computational burden on provers (who are already spending required to spend significant computational power to compute proofs).

Implementing query access. Section 3.5 introduced a straightforward way to implement efficient and secure query access to π' via a secure data availability service (e.g., Ethereum blob data). Both on- and off-chain, there may be other, more efficient approaches for realizing this access (or removing it entirely) and/or reducing the underlying trust assumptions.

Chapter 4

Cryptocurrency Mixers†

Contents

Bitcoin and cryptocurrencies sharing Bitcoin's core principles have attained huge prominence as decentralized and publicly verifiable payment systems. They

† *Portions of this section have been adapted from [GMM+22].*

have attracted not only cryptocurrency enthusiasts but also banks [Eur21], leading IT companies (e.g., Facebook and PayPal), and payment providers such as Visa [CEG+21]. At the same time, the initial perception of payment unlinkability based on pseudonyms has been refuted in numerous academic research works [MPJ+13, SHMSM21], and the blockchain surveillance industry [HSRK21] demonstrates this privacy breach in practice. This has led to a large amount of work devoted to providing a privacy-preserving overlay to Bitcoin in the form of *coin mixing* protocols [BBSU12, GFW22].

Decentralized coin mixing protocols such as CoinJoin [Wik22] or CoinShuffle [RMK14, RMK16, RM17] allow a set of mutually distrusting users to mix their coins to achieve *unlinkability*: that is, the coins cannot be linked to their initial owners even by malicious participants. These protocols suffer from a common drawback, the *bootstrapping problem*, i.e., how to find a set of participants to execute the protocol. In fact, while a high number of participants is desirable to improve the anonymity guarantees provided by the coin mixing protocol, such a high number is at the same time undesirable as it results in poor scalability and makes bootstrapping harder.

An alternative mechanism is one in which a third party, referred to as the *hub*, alleviates the bootstrapping problem by connecting users that want to mix their coins. Moreover, the hub itself can provide a coin mixing service by acting as a tumbler. In more detail, users send their coins to the hub, which, after collecting all the coins, sends them back to the users in a randomized order, thereby providing unlinkability for an observer of such transfers (e.g., an observer of the corresponding Bitcoin transactions).

Synchronization Puzzles There are numerous reported cases of "exit scams" by mixing services which took in new payments but stopped providing the mixing service [Sto22]. This has prompted the design of numerous cryptographic protocols [BNM+14, VR15, Coi22, HBG16] to remove trust from the hub, providing a trade-off between trust assumptions, minimum number of transactions, and Bitcoin compatibility [HAB+17]. Of particular interest is the work by Heilman et al. [HAB+17], which lays the groundwork for the core cryptographic primitive which can be used to build a mixing service. This primitive, referred to as a *synchronization puzzle*, enables unlinkability from even the view of a corrupt hub. However, Heilman et al. only present informal descriptions of the security and privacy notions of interest. Furthermore, the protocol proposed (TumbleBit) relies on hashed time-lock contracts (HTLCs), a smart contract incompatible with major cryptocurrencies such as Monero, Stellar, Ripple, MimbleWimble, and Zerocash (shielded addresses), lowering the interoperability of the solution.

The recent work of Tairi et al. [TMM21] attempts to overcome both of these limitations. It gives formal security notions for a synchronization puzzle in the universal composability (UC) framework [Can20]. It also provides an instantiation of the synchronization puzzle (called A^2L) that is simultaneously more efficient and more interoperable than TumbleBit, requiring only timelocks and digital signature verification from the underlying cryptocurrencies.

In this work, we identify a gap in their security analysis, and we substantiate the issue by presenting two concrete counterexamples: there exist two encryption schemes (secure under standard cryptographic assumptions) that satisfy the prerequisites of their security notions, yet yield completely insecure systems. This shows that our understanding of synchronization puzzles as a cryptographic primitive is still inadequate. Establishing firm foundations for this important cryptographic primitive requires us to rethink this object from the ground up.

4.1 Our Contributions

We summarize the contributions of this work below.

Counterexamples. First, we identify a gap in the security model of the synchronization puzzle protocol $\mathrm{A}^2\mathrm{L}$ [TMM21], presenting two concrete counterexamples (Section 4.6). Specifically, we show that there exist underlying cryptographic building blocks that satisfy the prerequisites stated in $\mathrm{A}^2\mathrm{L}$, yet they allow for:

- a *key recovery attack*, in which a user can learn the long-term secret decryption key of the hub;
- a *one-more signature attack*, in which a user can obtain n signed transactions from the hub while only engaging in $n-1$ successful instances of signing a transaction which pays the hub. In other words, the user obtains n coins from the hub while the hub receives only $n-1$ coins.

Both attacks run in polynomial time and succeed with overwhelming probability.

Definitions. To place the synchronization puzzle on firmer foundations, we propose a new cryptographic notion that we call *blind conditional signatures (BCS)*. Our new notion intuitively captures the functionality of a *synchronization puzzle* from [HAB$^+$17, TMM21]. BCS is a simple and easy-to-understand tool, and we formalize its security notions both in the *game-based* (Section 4.7) and *universal composability* (Section 4.9) setting. The proposed game-based definitions for BCS are akin to the well-understood standard security notions for regular blind signatures [Cha82, SU17]. We hope that this abstraction may lay the foundations for further studies on this primitive in all cryptocurrencies, scriptless or not.

Constructions. We give two constructions, one that satisfies our game-based security guarantees and one that is UC-secure. Both require only the same limited functionality as $\mathrm{A}^2\mathrm{L}$ from the underlying blockchain. In more detail:

- We give a modified version of $\mathrm{A}^2\mathrm{L}$ (Sections 4.8 and 4.8.1) which we refer to as $\mathrm{A}^2\mathrm{L}^+$ that satisfies the game-based notions (Section 4.7) of BCS, albeit in the *linear-only encryption (LOE)* model [Gro04]. In this model, the attacker does not directly have access to a homomorphic encryption scheme; instead, it can perform the legal operations by querying the corresponding oracles. This is a strong model with a non-falsifiable flavor, similar to the generic/algebraic group model [Sho97, Mau05, FKL18].

- We then provide a less efficient construction A^2L^{UC} that securely realizes the UC notion of BCS (Section 4.9). This scheme significantly departs from the construction paradigm of A^2L and is based on general-purpose cryptographic tools such as secure two-party computation (2PC).

Our results hint at the fact that achieving UC-security for a synchronization puzzle requires a radical departure from current construction paradigms, and it is likely to lead to less efficient schemes. On the other hand, we view the game-based definitions (a central contribution of our work) as a reasonable middle ground between security and efficiency.

4.2 Technical Overview

To put our work into context, we give a brief overview of A^2L [TMM21] recast as a synchronization puzzle (a notion first introduced in [HAB$^+$17]), and discuss how it can be used as a coin mixing protocol. We then outline the vulnerabilities in A^2L and discuss how to fix them using the tools that we develop in this work.

Synchronization Puzzles A synchronization puzzle protocol is a protocol between three parties: Alice, Bob, and Hub (refer to Figure 4.1 for a depiction). The synchronization puzzle begins with Hub and Bob executing a *puzzle promise* protocol (step 1) with respect to some message, m_{HB} such that Bob receives a puzzle τ that contains a signature s (at this point still hidden) on m_{HB}. Bob wishes to solve the puzzle and obtain the embedded signature. To do this, he sends the puzzle τ privately to Alice (step 2), who executes a *puzzle solve* protocol (step 3) with Hub with respect to some message m_{AH} such that, at the end of the protocol, Alice obtains the signature s, whereas Hub obtains a signature s' on m_{AH}. Alice then sends the signature s privately to Bob (step 4). Such a protocol must satisfy the following properties.

Blindness: The puzzle solve protocol does not leak any information to Hub about τ, and Hub *blindly* helps solve the puzzle. This ensures that Hub cannot link puzzles across interactions.

Unlockability: If step 3 is successfully completed, then the secret s must be a valid secret for Bob's puzzle τ. This guarantees that Hub cannot learn a signature on m_{AH}, without at the same time revealing a signature on m_{HB}.

Unforgeability: Bob cannot output a valid signature on m_{HB} before Alice interacts with the Hub.

Towards a Coin Mixing Service As shown in [HAB$^+$17, TMM21], the synchronization puzzle is the cryptographic core of a coin mixing service. First,

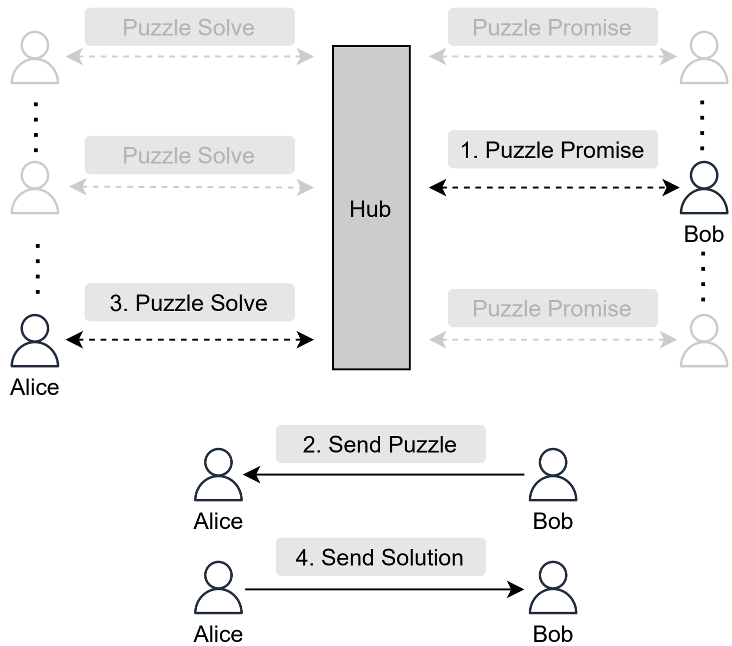

Figure 4.1: Protocol flow of the synchronization puzzle, the underlying cryptographic mechanism of Tumblebit and A^{2}L. Our approach in Blind Conditional Signatures follows a similar execution. Dotted double-edged arrows indicate 2-party protocols. Solid arrows indicate secure point-to-point communication.

Alice and Bob define the messages

$$m_{AH} : (A \xrightarrow{v} H) \text{ and } m_{HB} : (H \xrightarrow{v} B)$$

where $(U_i \xrightarrow{v} U_j)$ denotes a cryptocurrency payment (e.g., on-chain transaction or a payment over payment channels) that transfers v coins from U_i to U_j. Second, Alice and Bob run the synchronization puzzle protocol with Hub to synchronize the two aforementioned transfers. Here, the signatures s and s' are the ones required to validate the transactions defined by m_{AH} and m_{HB}. The anonymity of mixing follows from the fact that multiple pairs of users are executing the synchronization puzzle simultaneously with Hub, and Hub cannot link its interaction on the left to the corresponding interaction on the right. Throughout the rest of this work, we mainly focus on the synchronization puzzle as a cryptographic primitive. The application of a coin mixing protocol follows as prescribed in prior works [HAB$^+$17, TMM21].

The A^{2}L System In A^{2}L, the *blindness* property is achieved by making use of a re-randomizable linearly homomorphic (CPA-secure) encryption. The puzzle τ contains a ciphertext $c \leftarrow \mathsf{Enc}(\mathsf{ek}_H, s)$ encrypting the signature s under the encryption key ek_H of Hub. During the puzzle solve step, Alice first re-randomizes the ciphertext (and the underlying plaintext)

$$c \xrightarrow{r} c' = \mathsf{Enc}(\mathsf{ek}_H, s + r)$$

with a random scalar r. Hub then decrypts c' to obtain $s + r$, which in turn reveals a signature s' on m_{AH}.[1] Alice can then strip off the re-randomization factor r and send s to Bob later in step 4. In the analysis, it is argued that the CPA-security of the encryption scheme ensures unforgeability, whereas the re-randomization process guarantees blindness. Unfortunately, we show in this work that this claim is flawed.

Counterexamples We observe that the encryption scheme is only CPA-secure, and the Hub is offering a decryption oracle in disguise. In these settings, the right notion of security is the stronger CCA-security, which accounts exactly for this scenario. However, CCA-security is at odds with blindness, since we require the scheme to be (i) linearly homomorphic and (ii) publicly re-randomizable.[2] We then substantiate this concern by showing two counterexamples. Specifically, we show that there exist two encryption schemes that satisfy the prerequisites spelled out by A^{2}L, but enable two concrete attacks against the protocol. Depending on the scheme, we can launch one of the following attacks:

- A key recovery attack that completely recovers the long-term secret key of the hub, i.e., the decryption key dk_H.
- A one-more signature attack that allows one to obtain $n + 1$ signatures on transactions from Hub to Bob, while only revealing n signatures on transactions from Alice to Hub. Effectively, this allows one to steal coins from the hub.

We stress that both these schemes are specifically crafted to make the protocol fail: their purpose is to highlight a gap in the security model of A^{2}L. As such, they do not imply that A^{2}L as implemented is insecure, although we cannot prove it secure either. For a detailed description of the attacks, we refer the reader to Section 4.6.1.

Can We Fix This? In light of our attacks, the natural question is whether we can establish formally rigorous security guarantees for the (appropriately patched) A^{2}L system. While it seems unlikely that A^{2}L can achieve UC-security (more discussion on this later), we investigate whether it satisfies some weaker, but still meaningful, notion of security. Our main observation here is that a weak notion of CCA-security for encryption schemes suffices to provide formal

[1] This is achieved via the notion of *adaptor signatures*, but for the sake of this overview we ignore the exact details of this aspect.

[2] It is well known that no encryption scheme that satisfies either of these properties can be CCA-secure.

guarantees for A^2L. This notion, which we refer to as *one-more CCA-security*, (roughly) states that it is hard to recover the plaintexts of n ciphertexts while querying a decryption oracle at most $n-1$ times. Importantly, this notion is, in principle, not in conflict with the homomorphism/re-randomization requirements, contrary to standard CCA-security.

Towards establishing a formal analysis of A^2L, we introduce the notion of blind conditional signatures (BCS) as the cryptographic cornerstone of a synchronization puzzle. We propose game-based definitions (Section 4.7) similar in spirit to the well-established security definitions of regular blind signatures [Cha82, SU17]. We then prove that A^2L^+, our appropriately modified version of A^2L, satisfies these definitions (Section 4.8). Our analysis comes with an important caveat: we analyze the security of our scheme in the *linear-only encryption model*. This is a model introduced by Groth [Gro04] that only models adversaries that are restricted to perform "legal" operations on ciphertexts, similarly to the generic/algebraic group model. While this is far from a complete analysis, it increases our confidence in the security of the system.[3]

UC-Security The next question that we set out to answer is whether we can construct a synchronization puzzle that satisfies the strong notion of UC-security. We do not know how to prove that A^2L (or A^2L^+) is secure under composition, which is why we prove A^2L^+ secure only in the game-based setting. The technical difficulty in proving UC-security is that blindness is unconditional, and we lack a "trapdoor mechanism" that allows the simulator to link adversarial sessions during simulation in the security analysis; the proof of UC-security in [TMM21] is flawed due to this same reason. Thus, in Section 4.9.2 we develop a different protocol (called A^2L^{UC}) that we can prove UC-secure in the standard model. The scheme relies on standard general-purpose cryptographic tools, such as 2PC, and incurs a significant increase in computation costs. We stress that we view this scheme as a proof-of-concept, and leave further improvements for practical efficiency as an open problem. We hope that the scheme will shed some light on the barriers that need to be overcome in order to construct a practically efficient UC-secure synchronization puzzle.

4.3 Related Work

We recall some relevant related work in the literature.

Unlinkable Transactions CoinJoin [Wik22], Coinshuffle [RMK14, RMK16, RM17], and Möbius [MM18] are coin mixing protocols that rely on interested

[3]We resort to the LOE model because of the seemingly inherent conflict between linear homomorphism and CCA-like security, both of which are needed for our application (in our setting, the adversary has access to something akin to a decryption oracle). Indeed, even proving that ElGamal encryption is CCA1-secure in the standard model is a long-standing open problem, and we believe that the A^2L approach would inherently hit this barrier without some additional assumption.

users coming together and making an on-chain transactions to mix their coins. These proposals suffer from the bootstrapping problem (users having to find other interested users for the mix) in addition to requiring custom scripting language support from the underlying currency and completing the mix with on-chain transactions. Perun [DEFM19] and mixEth [SNBB19] are mixing solutions that rely on Ethereum smart contracts to resolve contentions among users. An alternate design choice is to incorporate coin unlinkability natively in the currency. Monero [LRR$^+$19] and Zcash [BCG$^+$14] are the two most popular examples of currencies that allow for unlinkable transactions without any special coin mixing protocol. This is enabled by complex on-chain cryptographic mechanisms that are not supported in other currencies.

RCCA Security A security notion related to one-more CCA is that of re-randomizable *Replayable CCA (RCCA)* encryption scheme [PR07]. The notion guarantees security even if the adversary has access to a decryption oracle, but only for ciphertexts that do not decrypt to the challenge messages. This is slightly different from what we require in our setting, since in our application the adversary will always query the oracle on encryption of new (non-challenge) messages (because of the plaintext re-randomization). This makes it challenging to leverage the guarantees provided by this notion in our analysis.

4.4 Additional Preliminaries

In this chapter, we require that any digital signature scheme $\Pi_{\mathsf{DS}} := (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ satisfies the standard notion of strong existential unforgeability [GMR88]. We require any non-interactive proof system NIZK (see Section 2.2) to be (1) *zero-knowledge*, i.e., there exists a simulator $\pi \leftarrow \mathsf{Sim}(\mathsf{td}, x)$ that computes valid proofs without the knowledge of the witness, (2) *sound*, i.e., it is infeasible for an adversary to output a valid proof for a statement $x \notin \mathcal{L}$ (see Definition 2), and (3) *UC-secure*, i.e., one can efficiently extract from the proofs computed by the adversary a valid witness (with the knowledge of the setup trapdoor td), even in the presence of simulated proofs. For formal security definitions, we refer the reader to [DMP88, CKS11].

In this chapter, we will deal with so-called *hard relations* $\mathcal{R}$ of statement-witness pairs (Y, y) which are decidable in polynomial time (see Section 2.2) but where, for all PPT adversaries $\mathcal{A}$, the probability of $\mathcal{A}$ on input Y outputting a witness y is negligible. We will also require a PPT sampling algorithm $\mathsf{GenR}(1^\lambda)$ that outputs a statement-witness pair $(Y, y) \in \mathcal{R}$. In particular, we use the discrete log relation $\mathcal{R}_{\mathsf{DL}}$ defined with respect to a group $\mathbb{G}$ with generator g and order p. The corresponding language is defined as $\mathcal{L}_{\mathsf{DL}} := \{Y : \exists y \in \mathbb{Z}_p,\ Y = g^y\}$.

4.4.1 Adaptor Signatures

Adaptor signatures [AEE+21a] let users generate a pre-signature on a message m which by itself is not a valid signature, but can later be adapted into a valid signature using knowledge of some secret value. More precisely, an adaptor signature scheme $\Pi_{\mathsf{ADP}} := (\mathsf{KGen}, \mathsf{PreSign}, \mathsf{PreVerify}, \mathsf{Adapt}, \mathsf{Verify}, \mathsf{Ext})$ is defined with respect to a signature scheme Π_{DS} and a hard relation $\mathcal{R}$. The key generation algorithm is the same as in Π_{DS} and outputs a key pair $(\mathsf{vk}, \mathsf{sk})$. The pre-signing algorithm $\mathsf{PreSign}(\mathsf{sk}, m, Y)$ returns a pre-signature $\tilde{\sigma}$ (we sometimes also refer to this as a partial signature). The pre-signature verification algorithm $\mathsf{PreVerify}(\mathsf{vk}, m, Y, \tilde{\sigma})$ verifies if the pre-signature $\tilde{\sigma}$ is correctly generated. The adapt algorithm $\mathsf{Adapt}(\tilde{\sigma}, y)$ transforms a pre-signature $\tilde{\sigma}$ into a valid signature σ given the witness y for the instance Y of the language $\mathcal{L}_{\mathcal{R}}$. The verification algorithm Verify is the same as in Π_{DS}. Finally, we have the extract algorithm $\mathsf{Ext}(\tilde{\sigma}, \sigma, Y)$ which, given a pre-signature $\tilde{\sigma}$, a signature σ, and an instance Y, outputs the witness y for Y. This can be formalized as *pre-signature correctness.*

Definition 9 (Pre-signature Correctness)**.** *An adaptor signature scheme* Π_{ADP} *satisfies pre-signature correctness if for every* $\lambda \in \mathbb{N}$*, every message* $m \in \{0,1\}^*$*, and every statement/witness pair* $(Y, y) \in \mathcal{R}$*, the following holds:*

$$\Pr\left[\begin{array}{c} \mathsf{PreVerify}(\mathsf{vk}, m, Y, \tilde{\sigma}) = 1 \\ \wedge \\ \mathsf{Verify}(\mathsf{vk}, m, \sigma) = 1 \\ \wedge \\ (Y, y') \in R \end{array} \middle| \begin{array}{l} (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda) \\ \tilde{\sigma} \leftarrow \mathsf{PreSign}(\mathsf{sk}, m, Y) \\ \sigma := \mathsf{Adapt}(\tilde{\sigma}, y) \\ y' := \mathsf{Ext}(\sigma, \tilde{\sigma}, Y) \end{array}\right] = 1.$$

In terms of security, we want standard unforgeability even when the adversary is given access to pre-signatures with respect to the signing key sk.

Definition 10 (Unforgeability)**.** *An adaptor signature scheme* Π_{ADP} *is* aEUF-CMA *secure if for every PPT adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{aSigForge}_{\mathcal{A},\Pi_{\mathsf{ADP}}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{aSigForge}_{\mathcal{A},\Pi_{\mathsf{ADP}}}$ *is defined as in Figure 4.2.*

We also require that, given a pre-signature and a witness for the instance, one can always adapt the pre-signature into a valid signature (*pre-signature adaptability*).

Definition 11 (Pre-signature Adaptability)**.** *An adaptor signature scheme* Π_{ADP} *satisfies pre-signature adaptability if for any* $\lambda \in \mathbb{N}$*, any message* $m \in \{0,1\}^*$*, any statement/witness pair* $(Y, y) \in R$*, any key pair* $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda)$*, and any pre-signature* $\tilde{\sigma} \leftarrow \{0,1\}^*$ *with* $\mathsf{PreVerify}(\mathsf{vk}, m, Y, \tilde{\sigma}) = 1$*, we have:*

$$\Pr[\mathsf{Verify}(\mathsf{vk}, m, \mathsf{Adapt}(\tilde{\sigma}, y)) = 1] = 1.$$

Finally, we require that, given a valid pre-signature and a signature with respect to the same instance, one can efficiently extract the corresponding witness

$\mathsf{aSigForge}_{\mathcal{A},\Pi_{\mathsf{ADP}}}(\lambda)$	$\mathcal{O}^{\mathsf{Sign}}(m)$
$\mathcal{Q} := \varnothing$	$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda)$	$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$m \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot),\mathcal{O}^{\mathsf{PreSign}}(\cdot,\cdot)}(\mathsf{vk})$	**return** σ
	$\mathcal{O}^{\mathsf{PreSign}}(m, Y)$
$(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$	$\tilde{\sigma} \leftarrow \mathsf{PreSign}(\mathsf{sk}, m, Y)$
$\tilde{\sigma} \leftarrow \mathsf{PreSign}(\mathsf{sk}, m, Y)$	$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot),\mathcal{O}^{\mathsf{PreSign}}(\cdot,\cdot)}(\tilde{\sigma}, Y)$	**return** $\tilde{\sigma}$
return $(m \notin \mathcal{Q} \wedge \mathsf{Verify}(\mathsf{vk}, m, \sigma))$	

Figure 4.2: Unforgeability experiment of adaptor signatures

(*witness extractability*).

Definition 12 (Witness Extractability). *An adaptor signature scheme* Π_{ADP} *is witness extractable if for every PPT adversary* $\mathcal{A}$*, there exists a negligible function* negl *such that*

$$\Pr[\mathsf{aWitExt}_{\mathcal{A},\Pi_{\mathsf{ADP}}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{aWitExt}_{\mathcal{A},\Pi_{\mathsf{ADP}}}$ *is defined as in Figure 4.3.*

$\mathsf{aWitExt}_{\mathcal{A},\Pi_{\mathsf{ADP}}}(\lambda)$	$\mathcal{O}^{\mathsf{Sign}}(m)$
$\mathcal{Q} := \varnothing$	$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KGen}(1^\lambda)$	$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$(m, Y) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot),\mathcal{O}^{\mathsf{PreSign}}(\cdot,\cdot)}(\mathsf{vk})$	**return** σ
$\tilde{\sigma} \leftarrow \mathsf{PreSign}(\mathsf{sk}, m, Y)$	$\mathcal{O}^{\mathsf{PreSign}}(m, Y)$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot),\mathcal{O}^{\mathsf{PreSign}}(\cdot,\cdot)}(\tilde{\sigma})$	$\tilde{\sigma} \leftarrow \mathsf{PreSign}(\mathsf{sk}, m, Y)$
$y' := \mathsf{Ext}(\sigma, \tilde{\sigma}, Y)$	$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(m \notin \mathcal{Q} \wedge (Y, y') \notin R$	**return** $\tilde{\sigma}$
$\wedge\, \mathsf{Verify}(\mathsf{vk}, m, \sigma))$	

Figure 4.3: Witness extractability experiment for adaptor signatures

Combining the three properties described above, we can define a secure adaptor signature scheme as follows.

Definition 13 (Secure Adaptor Signature Scheme). *An adaptor signature scheme* Π_{ADP} *is secure if it is* $\mathsf{aEUF\text{-}CMA}$ *secure, pre-signature adaptable, and witness extractable.*

$\mathcal{O}^{\mathsf{KGen}}(i)$	$\mathcal{O}^{\mathsf{Enc}}(\mathsf{ek}_i, m)$
$\mathsf{ek}_i \leftarrow\!\!\$\ \{0,1\}^\lambda$	$c_j \leftarrow\!\!\$\ \{0,1\}^\lambda$
Enter (i, ek_i) into table K	Enter (m, c_j) into table M_i
return ek_i	**return** c_j

$\mathcal{O}^{\mathsf{Dec}}(\mathsf{ek}_i, c)$

if $(\cdot, c) \notin M_i$ **then return** $\perp$
else
  Look up m corresponding to c in M_i
  return m

$\mathcal{O}^{\mathsf{Add}}(\mathsf{ek}_i, c_0, c_1)$

look up m_0, m_1 corresponding to c_0, c_1 in table M_i
$\tilde{c} \leftarrow\!\!\$\ \{0,1\}^\lambda$
Enter $(m_0 + m_1, \tilde{c})$ into table M_i
return $\tilde{c}$

Figure 4.4: Linear-only encryption oracles

4.4.2 Linear-Only Homomorphic Encryption

A public-key encryption scheme $\Pi_{\mathsf{E}} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ allows one to generate a key pair $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KGen}(1^\lambda)$ that allows anyone to encrypt messages as $c \leftarrow \mathsf{Enc}(\mathsf{ek}, m)$ and allows only the owner of the decryption key dk to decrypt ciphertexts as $m \leftarrow \mathsf{Dec}(\mathsf{dk}, c)$. We require that Π_{E} satisfies perfect correctness and the standard notion of CPA-security [GM82]. We say that an encryption scheme is *linearly homomorphic* if there exists some efficiently computable operation $\circ$ such that $\mathsf{Enc}(\mathsf{ek}, m_0) \circ \mathsf{Enc}(\mathsf{ek}, m_1) \in \mathsf{Enc}(\mathsf{ek}, m_0 + m_1)$, where addition is defined over $\mathbb{Z}_p$. The α-fold application of $\circ$ is denoted by $\mathsf{Enc}(\mathsf{ek}, m)^\alpha$.

Linear-only encryption (LOE) is an idealized model introduced by Groth [Gro04] as "generic homomorphic cryptosystem". Here, homomorphic encryption is modeled by giving access to oracles instead of their corresponding algorithms. A formal description of the oracles is given in Figure 4.4. We note that although we do not model such an algorithm explicitly, this model allows for (perfect) ciphertext re-randomization by homomorphically adding 0 to the desired ciphertext.

4.4.3 One-More Discrete Logarithm Assumption

We recall the one-more discrete logarithm (OMDL) assumption [BNPS03, BFP21].

Definition 14 (One-More Discrete Logarithm (OMDL) Assumption)**.** *Let $\mathbb{G}$ be a uniformly sampled cyclic group of prime order p and g a random generator* *of $\mathbb{G}$. The one-more discrete logarithm (OMDL) assumption states that for all $\lambda \in \mathbb{N}$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all PPT adversaries $\mathcal{A}$ making at most $q = \mathsf{poly}(\lambda)$ queries to $\mathsf{DL}(\cdot)$, the following holds:*

$$\Pr\left[\forall i : x_i = r_i \;\middle|\; \begin{array}{c} r_1, \ldots, r_{q+1} \leftarrow\!\!\$\ \mathbb{Z}_p \\ \forall i \in [q+1], h_i \leftarrow g^{r_i} \\ \{x_i\}_{i \in [q+1]} \leftarrow \mathcal{A}^{\mathsf{DL}(\cdot)}(h_1, \ldots, h_{q+1}) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where the $\mathsf{DL}(\cdot)$ oracle takes as input an element $h \in \mathbb{G}$ and returns x such that $h = g^x$.

4.5 The A^{2}L Protocol

We now recall the A^{2}L system [TMM21], which is defined over the following cryptographic schemes:

- A digital signature scheme Π_{DS}, a hard relation $\mathcal{R}_{\mathsf{DL}}$ for a group $(\mathbb{G}, g, p)$ with generator g and prime order p, and the corresponding adaptor signature scheme Π_{ADP}.
- A linearly homomorphic re-randomizable CPA-secure encryption scheme Π_{E}.[4]
- A NIZK proof system $\Pi_{\mathsf{NIZK}} := (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ for the language

$$\mathcal{L} := \{(\mathsf{ek}, Y, c) : \exists s \text{ s.t. } c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}, s) \wedge Y = g^s\}.$$

The protocol has three parties: Alice, Bob, and Hub. At the beginning of the system, Hub runs the setup (as described in Figure 4.13) to generate its keys, which are the keys for the (CPA-secure) encryption scheme Π_{E}. The protocol then consists of a promise phase and a solving phase. To conclude, Alice runs the open algorithm given in fig. 4.7.

Puzzle Promise In the promise phase (Figure 4.5), Hub generates a pre-signature $\tilde{\sigma}^H_{HB}$ on a common message m_{HB} with respect to a uniformly sampled instance $Y := g^s$. Hub also encrypts the witness s in the ciphertext $c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, s)$ under its own encryption key ek_H. Hub gives Bob the tuple $(Y, c, \pi, \tilde{\sigma}^H_{HB})$, where π is a NIZK proof that certifies the ciphertext c encrypts s. Bob verifies that the NIZK proof and the pre-signature are indeed valid. If so, he chooses a random $r \leftarrow\!\!\$\ \mathbb{Z}_q$ and re-randomizes the instance Y to $Y' := Y \cdot g^r$ and also re-randomizes the ciphertext c as $c' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Rand}(c, r)$. The puzzle is set to $\tau := (r, m_{HB}, \tilde{\sigma}^H_{HB}, (Y, c), (Y', c'))$.

[4]Technically, [TMM21] uses a different abstraction called "randomizable puzzle". However, it is not hard to see that a re-randomizable linearly homomorphic encryption scheme satisfies this notion. For completeness, we show this in Section 4.5.1.

Public parameters: group description $(\mathbb{G}, g, q)$, message m_{HB}

$\mathsf{PPromise}\langle H(\mathsf{dk}_H, \mathsf{sk}^H_{HB}), \cdot\rangle$		$\mathsf{PPromise}\langle \cdot, B(\mathsf{ek}_H, \mathsf{vk}^H_{HB})\rangle$
$s \leftarrow\!\!\$\ \mathcal{Z}_p, Y := g^s$		
$c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, s)$		
$\pi_s \leftarrow \mathsf{NIZK.Prove}((\mathsf{ek}_H, Y, c), s)$		
$\tilde{\sigma}^H_{HB} \leftarrow \Pi_{\mathsf{ADP}}.\tilde{\sigma}(\mathsf{sk}^H_{HB}, m_{HB}, Y)$		
	$\xrightarrow{Y, c, \pi_s, \tilde{\sigma}^H_{HB}}$	
		If $\mathsf{NIZK.Verify}((\mathsf{ek}_H, Y, c), \pi_s) \neq 1$ **then return** $\perp$
		If $\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}^H_{HB}, m_{HB}, Y, \tilde{\sigma}^H_{HB}) \neq 1$ **then**
		return $\perp$
		$r \leftarrow\!\!\$\ \mathcal{Z}_q, Y' := Y \cdot g^r$
		$c' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Rand}(c, r)$
		Set $\tau := (r, m_{HB}, \tilde{\sigma}^H_{HB}, (Y, c), (Y', c'))$
return $\perp$		**return** τ

Figure 4.5: Puzzle promise protocol of A^{2}L

Puzzle Solve Bob sends the puzzle τ privately to Alice, who now executes the puzzle solve protocol with Hub (Figure 4.6). Alice samples a random r' and further re-randomizes the instance Y' as $Y'' := Y' \cdot g^{r'}$ and the ciphertext c' as $c'' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Rand}(c', r')$. She then generates a pre-signature $\tilde{\sigma}^A_{AH}$ on a common message m_{AH} with respect to the instance Y''. She sends the tuple $(Y'', c'', \tilde{\sigma}^A_{AH})$ to Hub, who decrypts c'' using the decryption key dk_H to obtain s''. Hub then adapts the pre-signature $\tilde{\sigma}^A_{AH}$ to σ^A_{AH} using s'' and ensures its validity. It then sends the signature σ^A_{AH} to Alice, who extracts the witness for Y'' as $s'' \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Ext}(\tilde{\sigma}^A_{AH}, \sigma^A_{AH}, Y'')$. Alice removes the re-randomization factor to obtain the solution $s' := s'' - r'$ for the instance Y'. Alice finally sends s' privately to Bob, who opens the puzzle τ by computing the witness $s := s' - r$ and adapting the pre-signature $\tilde{\sigma}^H_{HB}$ (given by Hub in the promise phase) to the signature σ^H_{HB}.

4.5.1 Randomizable Puzzles and Homomorphic Encryption

Here we recall the definitions of randomizable puzzles [TMM21] and we show that they are trivially satisfied by a CPA-secure homomorphic encryption sceme (over $\mathbb{Z}_p$), with statistical circuit privacy [OPP14]. We recall the syntax as defined in [TMM21].

Definition 15 (Randomizable Puzzle)**.** *A randomizable puzzle scheme* $\mathsf{RP} = (\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PSolve}, \mathsf{PRand})$ *with a solution space* $\mathcal{S}$ *(and a function* ϕ *acting on* $\mathcal{S}$*) consists of four algorithms defined as:*

$(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{PSetup}(1^\lambda)$**:** *is a PPT algorithm that on input security parameter* 1^λ*, outputs public parameters* pp *and a trapdoor* td*.*

Public parameters: group description $(\mathbb{G}, g, q)$, message m_{AH}

$\mathsf{PSolve}\langle A(\mathsf{sk}^A_{AH}, \mathsf{ek}_H, \tau), \cdot\rangle$		$\mathsf{PSolve}\langle \cdot, H(\mathsf{dk}_H, \mathsf{vk}^A_{AH})\rangle$
Parse $\tau := (\cdot, \cdot, \cdot, \cdot, (Y', c'))$		
$r' \leftarrow\!\!\$\ \mathcal{Z}_p, Y'' := Y' \cdot g^{r'}$		
$c'' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Rand}(c', r')$		
$\tilde{\sigma}^A_{AH} \leftarrow \Pi_{\mathsf{ADP}}.\tilde{\sigma}(\mathsf{sk}^A_{AH}, m_{AH}, Y'')$		
	$\xrightarrow{Y'', c'', \tilde{\sigma}^A_{AH}}$	
		$s'' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c'')$
		$\sigma^A_{AH} \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}^A_{AH}, s'')$
		If $\Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}^A_{AH}, m_{AH}, \sigma^A_{AH}) \neq 1$ **then**
		return $\perp$
	$\xleftarrow{\sigma^A_{AH}}$	
$s'' \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Ext}(\sigma^A_{AH}, \tilde{\sigma}^A_{AH}, Y'')$		
If $s'' = \perp$ **then return** $\perp$		
$s' := s'' - r'$		
return (σ^A_{AH}, s')		**return** σ^A_{AH}

Figure 4.6: Puzzle solver protocol of A²L

$\mathsf{Open}(\tau, s')$

Parse $\tau := (r, \cdot, \tilde{\sigma}, \cdot, \cdot)$
$s := s' - r$
$\sigma \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}, s)$
return σ

Figure 4.7: Open algorithm of A²L

$Z \leftarrow \mathsf{PGen}(\mathsf{pp}, \zeta)$**:** *is a PPT algorithm that on input public parameters* pp *and a puzzle solution* ζ*, outputs a puzzle* Z*.*

$\zeta := \mathsf{PSolve}(\mathsf{td}, Z)$**:** *is a deterministic polynomial-time algorithm that on input a trapdoor* td *and puzzle* Z*, outputs a puzzle solution* ζ*.*

$(Z', r) \leftarrow \mathsf{PRand}(\mathsf{pp}, Z)$**:** *is a PPT algorithm that on input public parameters* pp *and a puzzle* Z *(which has a solution* ζ*), outputs a randomization factor* r *and a randomized puzzle* Z' *(which has a solution* $\phi(\zeta, r)$*).*

It is not hard to see that a linearly homomorphic encryption scheme (KGen, $\mathsf{Enc}, \mathsf{Dec}$) matches the syntax of a randomizable puzzle, setting pp to the encryption key and td to be the decryption key. For the PRand algorithm, we can sample a random $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and compute

$$\mathsf{Enc}(\mathsf{ek}, \zeta) \circ \mathsf{Enc}(\mathsf{ek}, r) = c$$

which is an encryption of $\phi(\zeta, r) = \zeta + r$. Next we recall the definition of security for randomizable puzzles.

Definition 16 (Security)**.** *A randomizable puzzle scheme* RP *is secure, if there exists a negligible function* negl*, such that*

$$\Pr\left[\zeta \leftarrow \mathcal{A}(\mathsf{pp}, Z) \;\middle|\; \begin{array}{l} (\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{PSetup}(1^\lambda) \\ \zeta \leftarrow\!\!\$\, \mathcal{S}, Z \leftarrow \mathsf{PGen}(\mathsf{pp}, \zeta) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

This follows as an immediate application of CPA-security (in fact, even the weaker one-wayness suffices) of the encryption scheme. Finally we recall the notion of privacy.

Definition 17 (Privacy)**.** *A randomizable puzzle scheme* RP *is private if for every PPT adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{RPRandSec}_{\mathcal{A},\mathsf{RP}}(\lambda) = 1] \leq 1/2 + \mathsf{negl}(\lambda)$$

where the experiment $\mathsf{RPRandSec}_{\mathcal{A},\mathsf{RP}}$ *is defined as follows:*

- $(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{PSetup}(1^\lambda)$
- $((Z_0, \zeta_0), (Z_1, \zeta_1)) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{td})$
- $b \leftarrow\!\!\$\, \{0, 1\}$
- $(Z'_0, r_0) \leftarrow \mathsf{PRand}(\mathsf{pp}, Z_0)$
- $(Z'_1, r_1) \leftarrow \mathsf{PRand}(\mathsf{pp}, Z_1)$
- $b' \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{td}, Z'_b)$
- *Return* $\mathsf{PSolve}(\mathsf{td}, Z_0) = \zeta_0 \wedge \mathsf{PSolve}(\mathsf{td}, Z_1) = \zeta_1$
  $\wedge\, b = b'$

Recall that circuit privacy implies that the distribution induced by $\mathsf{Enc}(\mathsf{ek}, \zeta) \circ \mathsf{Enc}(\mathsf{ek}, r)$ is statistically close to that induced by a a fresh encryption $\mathsf{Enc}(\mathsf{ek}, \zeta + r)$. This implies that privacy is satisfied in a statistical sense. Thus we can state the following.

Lemma 2. *Assuming that* $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is a linearly homomorphic encryption with statistical circuit privacy, the there exists a randomizable puzzle with statistical privacy.*

4.6 Counterexamples of A^{2}L

Next, we describe two cryptographic instantiations of A^{2}L that satisfy the formal definitions, yet enable two attacks. For the purpose of these attacks, it suffices to keep in mind that Hub offers the sender party (Alice) access to the following oracle, which we refer to as $\mathcal{O}^{\mathsf{A^2L}}_{\mathsf{dk},\Pi_\mathsf{E},\Pi_\mathsf{ADP}}$. On input a verification key vk, a message m, a group element h, a ciphertext c, and a partial signature $\tilde{\sigma}$, the oracle behaves as follows:

- Compute $\tilde{x} \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}, c)$.
- Compute $\sigma' \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}, \tilde{x})$.
- If $\Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}, m, \sigma') = 1$, return σ'.
- Else return $\perp$.

Note that returning σ' implicitly reveals $\tilde{x}$, since $\Pi_{\mathsf{ADP}}.\mathsf{Ext}(\tilde{\sigma}, \sigma', h) = \tilde{x}$. It is also useful to observe that providing a valid pre-signature to the A^{2}L oracle is trivial for an adversary: generating a pre-signature that is valid when adapted with a value x requires only knowledge of the party's own signing key and of a value $h = g^x$. The leakage offered by this oracle (and indeed the existence of this leakage) is not addressed in A^{2}L's proof of security.

4.6.1 Key Recovery Attack

In our first attack, we completely recover the decryption key dk of the hub by simply querying the oracle $\mathcal{O}^{\mathsf{A^2L}}_{\mathsf{sk},\Pi_{\mathsf{E}},\Pi_{\mathsf{ADP}}}$ n times. For this attack, we assume that the encryption scheme Π_{E} is (in addition to being re-randomizable and CPA-secure as required by A^{2}L):

- Linearly homomorphic over $\mathbb{Z}_p$.
- Circular secure for bit encryption, i.e., the scheme is CPA-secure even given the bitwise encryption of the decryption key $\mathsf{Enc}(\mathsf{ek}, \mathsf{dk}_1), \ldots, \mathsf{Enc}(\mathsf{ek}, \mathsf{dk}_\lambda)$.
- The above-mentioned ciphertexts $(c_1, \ldots, c_\lambda) := (\mathsf{Enc}(\mathsf{ek}, \mathsf{dk}_1), \ldots, \mathsf{Enc}(\mathsf{ek}, \mathsf{dk}_\lambda))$ are included in the encryption key ek.

Such schemes can be constructed from a variety of standard assumptions [BHHO08]. It is easy to see that these additional requirements do not contradict the initial prerequisites of the scheme.

The attack is shown in Algorithm 1. Note that, for a signing key pair in the i-th iteration, if the $\mathcal{O}^{\mathsf{A^2L}}$ oracle returns $y \neq \perp$, this means that in the coin mixing layer, the Hub has obtained a valid y and thus obtains Alice's (adversary's) signature on a transaction. Due to one-time use of keys in this (cryptocurrency) layer, the attacker therefore cannot reuse the same signing key pair in another iteration for a different message (transaction). Therefore, it is necessary that the attacker (Alice) sample n signing keys to account for every iteration being a non-$\perp$ query to $\mathcal{O}^{\mathsf{A^2L}}$. This is realized in the real world by the attacker having n different sessions (of coin mixing), one for each vk_i, with Hub.

Observe that the response of the oracle is $\perp$ if and only if $\mathsf{dk}_i = 1$, since $h = g^x \neq g^{x+1}$. On the other hand, if $\mathsf{dk}_i = 0$, then the oracle always returns a valid adapted signature σ'. Thus, the attack succeeds with probability 1.

Algorithm 1 Key Recovery Attack

Input: Hub's ek along with the cipheretexts $(c_1, \ldots, c_\lambda)$

Initialize key guess $\mathsf{dk}' := 0^\lambda$

for $i \in 1 \ldots \lambda$ **do**

  Sample $x \leftarrow\$\ \mathbb{Z}_p$ and compute $h := g^x$

  Sample a fresh signing key $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$

  Set $c'_i := \Pi_\mathsf{E}.\mathsf{Enc}(\mathsf{ek}, x) \circ c_i = \Pi_\mathsf{E}.\mathsf{Enc}(\mathsf{ek}, x + \mathsf{dk}_i)$

  Compute $\tilde{\sigma}_i \leftarrow \Pi_\mathsf{ADP}.\mathsf{PreSign}(\mathsf{sk}, m, h)$

  Query $y \leftarrow \mathcal{O}^{\mathsf{A^2L}}_{\mathsf{dk},\Pi_\mathsf{E},\Pi_\mathsf{ADP}}(\mathsf{vk}, m, h, c'_i, \tilde{\sigma}_i)$

  If $y = \perp$ set $\mathsf{dk}'_i := 1$

end for

return dk'

4.6.2 One-More Signature Attack

We present a different attack, where we impose different assumptions on the encryption scheme Π_E. We discuss later in the section why these assumptions do not contradict the pre-requisites of the A^{2}L scheme. Specifically, in addition to A^{2}L's requirement that the scheme is perfectly re-randomizable and CPA-secure, we assume that it is:

- Linearly homomorphic over $\mathbb{Z}_p$.
- Supports homomorphic evaluation of the *conditional bit flip* (CFlip) function, defined as

$$\Pi_\mathsf{E}.\mathsf{CFlip}(\mathsf{ek}, i, \mathsf{Enc}(\mathsf{ek}, x)) := \mathsf{Enc}(\mathsf{ek}, y)$$
$$\text{where } \begin{cases} y = x & \text{if } x_i = 0 \\ y = x \oplus e_i & \text{if } x_i = 1 \end{cases}$$

  and e_i is the i-th unit vector.

The objective of the attack is to steal coins from the hub in the coin mixing protocol. Specifically, at the A^{2}L level, the attacker will solve $q + 1$ puzzles by querying the puzzle solver interface successfully only q times. Note that we do not count unsuccessful (i.e., the oracle returns $\perp$) queries, since those non-accepting queries do not correspond to any payment from Alice's side.

The attack is shown in Algorithm 2. We assume (for convenience) that $q \geq \lambda$ and that $\mathbb{Z}_p \leq 2^\lambda$ and therefore $x_j \in \{0,1\}^\lambda$. Observe that the attack makes at most q successful queries to the oracle, so all we need to show is that the success probability is high enough. First, we argue that the attack recovers the correct $x'_1 = x_1$ with probability 1. If the i-th bit $x_{1,i} = 0$, then the CFlip operation does not alter the content of the ciphertext and therefore

$$c' = \mathsf{Enc}\left(\mathsf{ek}, \sum_{j=1}^{q+1} r_j^{(i)} \cdot x_j\right) \text{ and } h' = \prod_{j=1}^{q+1} h_j^{r_j^{(i)}} = g^{\sum_{j=1}^{q+1} r_j^{(i)} \cdot x_j}$$

Algorithm 2 One-More Signature Attack

Input: Bob's ciphertexts $(c_1, \ldots, c_{q+1})$ and group elements $(h_1, \ldots, h_{q+1})$, where $c_j = \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}, x_j)$ and $h_j := g^{x_j}$, and Hub's ek

Initialize guess $x'_1 := 0^\lambda$ and a counter $i := 1$

for $i = 1 \ldots \lambda$ **do**

  Sample a fresh signing key $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$

  Compute $c'_1 \leftarrow \Pi_{\mathsf{E}}.\mathsf{CFlip}(\mathsf{ek}, i, c_1)$

  Sample $(r_1^{(i)}, \ldots, r_{q+1}^{(i)}) \leftarrow\$ \mathbb{Z}_p^{q+1}$

  Compute $c' := (c'_1)^{r_1^{(i)}} \circ (c_2)^{r_2^{(i)}} \cdots \circ (c_{q+1})^{r_{q+1}^{(i)}}$

  Compute $h' := \prod_{j=1}^{q+1} h_j^{r_j^{(i)}}$

  Sign $\tilde{\sigma} \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{PreSign}(\mathsf{sk}, m, h')$

  Query $y_i \leftarrow \mathcal{O}^{\mathsf{A^2L}}_{\mathsf{dk}, \Pi_{\mathsf{E}}, \Pi_{\mathsf{ADP}}}(\mathsf{vk}, m, h', c', \tilde{\sigma})$

  If $y_i = \perp$ set $x'_{1,i} := 1$

end for

Continue querying (without updating x'_1) until q non-$\perp$ queries have been made

For all i corresponding to a non-$\perp$ query, set E_i to be the equation $y_i - r_1^{(i)} x'_1 = r_2^{(i)} x'_2 + \ldots + r_{q+1}^{(i)} x'_{q+1}$

Solve $(E_1, \ldots, E_q)$ for $(x'_2, \ldots, x'_{q+1})$

return $(x'_1, x'_2, \ldots, x'_{q+1})$

so the oracle always returns a non-$\perp$ response. On the other hand, if $x_{1,i} = 1$, then the above equality does not hold and therefore $\mathcal{O}^{\mathsf{A^2L}}_{\mathsf{sk}, \Pi_{\mathsf{E}}, \Pi_{\mathsf{ADP}}}$ always returns $\perp$.

This querying strategy is repeated for every bit of x'_1 and continued on x_2, etc., until q non-$\perp$ queries have been made. Because $q \geq \lambda$, the attacker will have learned all λ bits of x'_1 by this point. Thus, the set of equations $(E_1, \ldots, E_q)$ has exactly q unknowns. Since the coefficients are uniformly chosen, the equations are, with all but negligible probability, linearly independent. Since $\mathbb{Z}_p$ is a field, the solution is uniquely determined and can be found efficiently via Gaussian elimination.

N-More Signatures The described attack is in fact even stronger than shown. Using this method, an attacker $\mathcal{A}$ can use q queries, where $\lfloor q \rfloor = N\lambda$, to recover $N + q$ plaintexts. $\mathcal{A}$ does this by using $N\lambda$ queries to recover the first N plaintexts $x_1, \ldots, x_N$ and $N\lambda$ equations as described previously (once it has flipped all λ bits in x_1, it starts flipping bits in x_2, and so on). Using its remaining queries, it obtains $q - N\lambda$ more equations (either by continuing to flip bits in further ciphertexts, which are however wasted, or by simply choosing new values r_i for the linear combinations) for a total of q equations. Using Gaussian elimination, it can recover the remaining q plaintexts $x_{N+1}, \ldots, x_{N+q}$. Taken with the plaintexts $x_1, \ldots, x_N$ that were recovered bit-by-bit, the attacker has learned $N + q$ plaintexts.

Instantiations We now justify our additional assumptions on the encryption scheme Π_{E} by describing suitable instantiations that satisfy all the requirements. Clearly, if the scheme is fully-homomorphic [Gen09] then it supports both linear functions over $\mathbb{Z}_p$ and conditional bit flips. However, we show that even a linear homomorphic encryption (over $\mathbb{Z}_p$) can suffice to mount our attack. Specifically, given a CPA-secure linearly homomorphic encryption scheme $(\mathsf{KGen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*)$, we define a *bitwise* encryption scheme $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ as follows:

- $\mathsf{KGen}(1^\lambda)$: Return the output of $\mathsf{KGen}^*(1^\lambda)$.
- $\mathsf{Enc}(\mathsf{ek}, x)$: Parse x as $(x^{(1)}, \ldots, x^{(n)})$ and return $(\mathsf{Enc}^*(\mathsf{ek}, x^{(1)}), \ldots, \mathsf{Enc}^*(\mathsf{ek}, x^{(n)}))$.
- $\mathsf{Dec}(\mathsf{dk}, c)$: Parse c as $(c^{(1)}, \ldots, c^{(\lambda)})$ and return $\sum_{i=1}^{\lambda} 2^{i-1} \cdot \mathsf{Dec}^*(\mathsf{dk}, c^{(i)})$.

It is easy to show that the new scheme is CPA-secure via a standard hybrid argument.

Next, we argue that one can efficiently implement the conditional bit flip operation (CFlip) over such ciphertexts. Given a ciphertext $c = (c^{(1)}, \ldots, c^{(\lambda)})$, we can conditionally flip the i-th bit by computing

$$(c^{(1)}, \ldots, \underbrace{\mathsf{Enc}^*(\mathsf{ek}, 0)}_{i\text{-th ciphertext}}, \ldots, c^{(\lambda)}).$$

This is a correctly formed ciphertext, since the conditional bit flip always sets the i-th bit to 0 and leaves the other positions untouched.

Finally, we need to argue that the encryption scheme is still linearly homomorphic over $\mathbb{Z}_p$. Note that this does not follow immediately from the fact that $(\mathsf{KGen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*)$ is linearly homomorphic, since the new encryption algorithm decomposes the inputs bitwise. Nevertheless, we show this indeed holds for the case of two ciphertexts $c = (c^{(1)}, \ldots, c^{(\lambda)})$ and $d = (d^{(1)}, \ldots, d^{(\lambda)})$ encrypting x and y, respectively. The general case follows analogously. To homomorphically compute $\alpha x + \beta y$, where $(\alpha, \beta) \in \mathbb{Z}_p^2$, we compute

$$\left(\left(\bigcirc_{i=1}^{\lambda} (c^{(i)})^{2^{i-1}} \right)^{\alpha} \circ \left(\bigcirc_{i=1}^{\lambda} (d^{(i)})^{2^{i-1}} \right)^{\beta}, \mathsf{Enc}^*(\mathsf{ek}, 0), \ldots, \mathsf{Enc}^*(\mathsf{ek}, 0) \right).$$

A routine calculation shows that this ciphertext correctly decrypts to the desired result $\alpha x + \beta y$.

4.7 Blind Conditional Signatures

Next, we formally define and instantiate blind conditional signatures, the central cryptographic notion for coin mixing services. Our goal here is to give a simple and easy-to-understand formalization of a synchronization puzzle.

A blind conditional signature (BCS) is executed among users Alice, Bob, and Hub. The interfaces and associated security properties are defined below.

Definition 18 (Blind Conditional Signature)**.** *A blind conditional signature* $\Pi_{\mathsf{BCS}} := (\mathsf{Setup}, \mathsf{PPromise}, \mathsf{PSolve}, \mathsf{Open})$ *is defined with respect to a signature scheme* $\Pi_{\mathsf{DS}} := (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ *and consists of the following efficient algorithms.*

- $(\tilde{\mathsf{ek}}, \tilde{\mathsf{dk}}) \leftarrow \mathsf{Setup}(1^\lambda)$*: The setup algorithm takes as input the security parameter* 1^λ *and outputs a key pair* $(\tilde{\mathsf{ek}}, \tilde{\mathsf{dk}})$*.*

- $(\perp, \{\tau, \perp\}) \leftarrow \mathsf{PPromise}\left\langle \begin{matrix} H\left(\tilde{\mathsf{dk}}, \mathsf{sk}^H, m_{HB}\right) \\ B\left(\tilde{\mathsf{ek}}, \mathsf{vk}^H, m_{HB}\right) \end{matrix} \right\rangle$*: The puzzle promise algorithm is an interactive protocol between two users* H *(with inputs the decryption key* $\tilde{\mathsf{dk}}$*, the signing key* sk^H*, and a message* m_{HB}*) and* B *(with inputs the encryption key* $\tilde{\mathsf{ek}}$*, the verification key* vk^H*, and a message* m_{HB}*) and returns* $\perp$ *to* H *and either a puzzle* τ *or* $\perp$ *to* B*.*

- $(\{(\sigma^*, s), \perp\}, \{\sigma^*, \perp\}) \leftarrow \mathsf{PSolve}\left\langle \begin{matrix} A\left(\mathsf{sk}^A, \tilde{\mathsf{ek}}, m_{AH}, \tau\right) \\ H\left(\tilde{\mathsf{dk}}, \mathsf{vk}^A, m_{AH}\right) \end{matrix} \right\rangle$*: The puzzle solving algorithm is an interactive protocol between two users* A *(with inputs the signing key* sk^A*, the encryption key* $\tilde{\mathsf{ek}}$*, a message* m_{AH}*, and a puzzle* τ*) and* H *(with inputs the decryption key* $\tilde{\mathsf{dk}}$*, the verification key* pk^A*, and a message* m_{AH}*) and returns to both users either a signature* σ^* *(*A *additionally receives a secret* s*) or* $\perp$*.*

- $\{\sigma, \perp\} \leftarrow \mathsf{Open}(\tau, s)$*: The open algorithm takes as input a puzzle* τ *and a secret* s *and returns a signature* σ *or* $\perp$*.*

Next, we define correctness.

Definition 19 (Correctness)**.** *A blind conditional signature* Π_{BCS} *is correct if for all* $\lambda \in \mathbb{N}$*, all* $(\tilde{\mathsf{ek}}, \tilde{\mathsf{dk}})$ *in the support of* $\mathsf{Setup}(1^\lambda)$*, all* $(\mathsf{vk}^H, \mathsf{sk}^H)$ *and* $(\mathsf{vk}^A, \mathsf{sk}^A)$ *in the support of* $\Pi_{\mathsf{DS}}.\mathsf{KGen}(1^\lambda)$*, and all pairs of messages* (m_{HB}, m_{AH})*, it holds that*

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}^H, m_{HB}, \mathsf{Open}(\tau, s)) = 1\right] = 1$$

and

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}^A, m_{AH}, \sigma^*) = 1\right] = 1$$

where

- $\tau \leftarrow \mathsf{PPromise}\left\langle \begin{matrix} H\left(\tilde{\mathsf{dk}}, \mathsf{sk}^H, m_{HB}\right) \\ B\left(\tilde{\mathsf{ek}}, \mathsf{vk}^H, m_{HB}\right) \end{matrix} \right\rangle$ *and*

- $((\sigma^*, s), \sigma^*) \leftarrow \mathsf{PSolve}\left\langle \begin{array}{l} A\left(\mathsf{sk}^A, \tilde{\mathsf{ek}}, m_{AH}, \tau\right) \\ H\left(\tilde{\mathsf{dk}}, \mathsf{vk}^A, m_{AH}\right) \end{array} \right\rangle$.

We now present the security guarantees of BCS in the game-based setting. Our definition of blindness is akin to the strong blindness notion of standard blind signatures [Cha82], in which the adversary picks the keys (as opposed to the weak version in which they are chosen by the experiment)[5]. Roughly speaking, it says that two promise/solve sessions cannot be linked together by the hub.[6]

Definition 20 (Blindness)**.** *A blind conditional signature Π_{BCS} is blind if there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$, the following holds:*

$$\Pr\Big[\mathsf{ExpBlnd}^{\mathcal{A}}_{\Pi_{\mathsf{puzzle}}}(\lambda) = 1\Big] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

where $\mathsf{ExpBlnd}$ *is defined in Figure 4.8.*[7]

Next, we define unlockability, which says that it should be hard for Hub to create a valid signature on Alice's message that does not allow Bob to unlock the full signature in the corresponding promise session.

Definition 21 (Unlockability)**.** *A blind conditional signature Π_{BCS} is unlockable if there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$, the following holds:*

$$\Pr\Big[\mathsf{ExpUnlock}^{\mathcal{A}}_{\Pi_{\mathsf{BCS}}}(\lambda) = 1\Big] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{ExpUnlock}$ *is defined in Figure 4.9.*

Our definition of unforgeability is inspired by the unforgeability of blind signatures [Cha82]. We require that Alice and Bob cannot recover q signatures from Hub while successfully querying the solving oracle at most $q-1$ times. Since each successful query reveals a signature from Alice's key (which in turn corresponds to a transaction from Alice to Hub), this requirement implicitly captures the fact that Alice and Bob cannot steal coins from Hub. The winning condition b_0 captures the scenario where the adversary forges a signature of the hub on a message previously not used in any promise oracle query. The remaining conditions b_1, b_2 and b_3 together capture the scenario in which the

[5] We opt for this stronger version since we want to provide anonymity even in the case of a fully malicious hub, which can pick its keys adversarially to try to link a sender/receiver pair.

[6] We do not consider the case in which Hub colludes with either Alice or Bob, since deanonymization is trivial (Alice (resp. Bob) simply reveals the identity of Bob (resp. Alice) to Hub); this is in line with [TMM21].

[7] In previous works, descriptions of unlinkability assume an explicit step for blinding the puzzle τ between $\mathsf{PPromise}$ and PSolve. Here, we assume that PSolve performs this blinding functionality.

$$
\begin{array}{|l|}
\hline
\mathsf{ExpBlnd}^{\mathcal{A}}_{\Pi_{\mathsf{BCS}}}(\lambda) \\
\hline
(\tilde{\mathsf{ek}}, \mathsf{vk}^H_0, \mathsf{vk}^H_1, (m_{HB,0}, m_{AH,0}), (m_{HB,1}, m_{AH,1})) \leftarrow \mathcal{A}(1^\lambda) \\
(\mathsf{vk}^A_0, \mathsf{sk}^A_0) \leftarrow \mathsf{KGen}(1^\lambda) \\
(\mathsf{vk}^A_1, \mathsf{sk}^A_1) \leftarrow \mathsf{KGen}(1^\lambda) \\
\tau_0 \leftarrow \mathsf{PPromise}\left\langle \mathcal{A}(\mathsf{vk}^A_0, \mathsf{vk}^A_1), B(\tilde{\mathsf{ek}}, \mathsf{vk}^H_0, m_{HB,0}) \right\rangle \\
\tau_1 \leftarrow \mathsf{PPromise}\left\langle \mathcal{A}(\mathsf{vk}^A_0, \mathsf{vk}^A_1), B(\tilde{\mathsf{ek}}, \mathsf{vk}^H_1, m_{HB,1}) \right\rangle \\
b \leftarrow \{0,1\} \\
(\sigma^*_0, s_0) \leftarrow \mathsf{PSolve}\left\langle A\left(\mathsf{sk}^A_0, \tilde{\mathsf{ek}}, m_{AH,0}, \tau_{0\oplus b}\right), \mathcal{A} \right\rangle \\
(\sigma^*_1, s_1) \leftarrow \mathsf{PSolve}\left\langle A\left(\mathsf{sk}^A_1, \tilde{\mathsf{ek}}, m_{AH,1}, \tau_{1\oplus b}\right), \mathcal{A} \right\rangle \\
\textbf{if } (\sigma^*_0 = \bot) \vee (\sigma^*_1 = \bot) \vee (\tau_0 = \bot) \vee (\tau_1 = \bot) \\
\quad \sigma_0 := \sigma_1 := \bot \\
\textbf{else} \\
\quad \sigma_{0\oplus b} \leftarrow \mathsf{Open}(\tau_{0\oplus b}, s_0) \\
\quad \sigma_{1\oplus b} \leftarrow \mathsf{Open}(\tau_{1\oplus b}, s_1) \\
b' \leftarrow \mathcal{A}(\sigma_0, \sigma_1) \\
\textbf{return } (b = b') \\
\hline
\end{array}
$$

Figure 4.8: Blindness experiment

adversary outputs q valid distinct key-message-signature tuples while having queried for solve only $q-1$ times. Hence, in the second condition, the attacker manages to *complete* q promise interactions with only $q-1$ solve interactions, whereas in the first winning condition, the adversary computes a *fresh* signature that is not the completion of any promise interaction. These conditions are technically incomparable: an attacker that succeeds under one condition does not imply an attacker succeeding on the other. It is important to note that this is different from the unforgeability notion of blind signatures (where the attacker only has access to a single signing oracle), since in our case the hub is offering the attacker two oracles: promise and solve.

Definition 22 (Unforgeability)**.** *A blind conditional signature* Π_{BCS} *is* unforgeable *if there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for all* $\lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$*, the following holds:*

$$\Pr\left[\mathsf{ExpUnforg}^{\mathcal{A}}_{\Pi_{\mathsf{BCS}}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

where ExpUnforg *is defined in Figure 4.10.*

We define security as the collection of all properties.

$\mathsf{ExpUnlock}^{\mathcal{A}}_{\Pi_{\mathsf{BCS}}}(\lambda)$

$(\tilde{\mathsf{ek}}, \mathsf{vk}^H, m_{HB}, m_{AH}) \leftarrow \mathcal{A}(1^\lambda)$
$(\mathsf{vk}^A, \mathsf{sk}^A) \leftarrow \mathsf{KGen}(1^\lambda)$
$\tau \leftarrow \mathsf{PPromise}\left\langle \mathcal{A}(\mathsf{vk}^A), B(\tilde{\mathsf{ek}}, \mathsf{vk}^H, m_{HB})\right\rangle$
if $\tau = \perp$
$\quad(\hat{\sigma}, \hat{m}) \leftarrow \mathcal{A}$
$\quad b_0 := (\mathsf{Verify}(\mathsf{vk}^A, \hat{\sigma}, \hat{m}) = 1)$
if $\tau \neq \perp$
$\quad(\sigma^*, s) \leftarrow \mathsf{PSolve}\left\langle A\left(\mathsf{sk}^A, \tilde{\mathsf{ek}}, m_{AH}, \tau\right), \mathcal{A}\right\rangle$
$\quad(\hat{\sigma}, \hat{m}) \leftarrow \mathcal{A}$
$\quad b_1 := (\mathsf{Verify}(\mathsf{vk}^A, \hat{\sigma}, \hat{m}) = 1) \wedge (\hat{m} \neq m_{AH})$
$\quad b_2 := (\mathsf{Verify}(\mathsf{vk}^A, \sigma^*, m_{AH}) = 1)$
$\quad b_3 := (\mathsf{Verify}(\mathsf{vk}^H, m_{HB}, \mathsf{Open}(\tau, s)) \neq 1)$
return $b_0 \vee b_1 \vee (b_2 \wedge b_3)$

Figure 4.9: Unlockability experiment

Definition 23 (Security)**.** *A blind conditional signature* Π_{BCS} *is secure if it is blind, unlockable, and unforgeable.*

4.8 The A^{2}L$^+$ Protocol

In the following we describe our A^{2}L$^+$ construction. Our scheme is a provable variant of A^{2}L (Section 4.5) and therefore we only describe the differences with respect to the original protocol. The concrete modifications are as follows:

- Augment the public key of Hub ek_H with a NIZK proof that certifies that $\mathsf{ek}_H \in \mathsf{Supp}(\Pi_{\mathsf{E}}.\mathsf{KGen}(1^\lambda))$. All parties verify this proof during their first interaction with Hub.
- In PSolve (Figure 4.6), Hub additionally checks if vk^A_{AH} is in the support of $\Pi_{\mathsf{ADP}}.\mathsf{KGen}(1^\lambda)$ before the decryption (line 6). Furthermore, we replace the condition (line 8) with

$$\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}^A_{AH}, m_{AH}, Y'', \tilde{\sigma}^A_{AH}) \neq 1 \vee \; g^{s''} \neq Y''.$$

4.8.1 Security Analysis

Before proving our main theorem, we define a property which is going to be useful for our analysis.

$$
\begin{array}{l}
\underline{\mathsf{ExpUnforg}^{\mathcal{A}}_{\Pi_{\mathsf{BCS}}}(\lambda)} \\
\mathcal{L} := \varnothing, Q := 0 \\
(\tilde{\mathsf{ek}}, \tilde{\mathsf{dk}}) \leftarrow \mathsf{Setup}(1^\lambda) \\
(\mathsf{vk}_1^H, m_1, \sigma_1), \ldots, (\mathsf{vk}_q^H, m_q, \sigma_q) \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{PP}(\cdot), \mathcal{O}\mathsf{PS}(\cdot)}(\tilde{\mathsf{ek}}) \\
b_0 := \exists i \in [q] \text{ s.t. } (\mathsf{vk}_i^H, \cdot) \in \mathcal{L} \wedge (\mathsf{vk}_i^H, m_i) \notin \mathcal{L} \\
\qquad\qquad \wedge\, \mathsf{Verify}(\mathsf{vk}_i^H, m_i, \sigma_i) = 1 \\
b_1 := \forall i \in [q], (\mathsf{vk}_i^H, m_i) \in \mathcal{L} \wedge \mathsf{Verify}(\mathsf{vk}_i^H, m_i, \sigma_i) = 1 \\
b_2 := \bigwedge_{i,j \in [q], i \neq j} (\mathsf{vk}_i^H, m_i, \sigma_i) \neq (\mathsf{vk}_j^H, m_j, \sigma_j) \\
b_3 := (Q \leq q - 1) \\
\textbf{return } b_0 \vee (b_1 \wedge b_2 \wedge b_3) \\
\\
\underline{\mathcal{O}\mathsf{PP}(m)} \\
(\mathsf{vk}^H, \mathsf{sk}^H) \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{KGen}(1^\lambda) \\
\mathcal{L} := \mathcal{L} \cup \{(\mathsf{vk}^H, m)\} \\
\perp \leftarrow \mathsf{PPromise}\langle H(\tilde{\mathsf{dk}}, \mathsf{sk}^H, m), \mathcal{A}(\mathsf{vk}^H) \rangle \\
\\
\underline{\mathcal{O}\mathsf{PS}(\mathsf{vk}^A, m')} \\
\sigma^* \leftarrow \mathsf{PSolve}\langle \mathcal{A}, H(\tilde{\mathsf{dk}}, \mathsf{vk}^A, m') \rangle \\
\textbf{if } \sigma^* \neq \perp \textbf{ then } Q := Q + 1
\end{array}
$$

Figure 4.10: Unforgeability experiment

Definition 24 (OM-CCA-A2L)**.** *An encryption scheme* Π_{E} *is one-more CCA-A2L-secure (OM-CCA-A2L) if there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for all* $\lambda \in \mathbb{N}$*, all polynomials* $q = q(\lambda)$*, and all PPT adversaries* $\mathcal{A}$*, the following holds:*

$$\Pr\left[\mathit{OM\text{-}CCA\text{-}A2L}^{\mathcal{A}}_{\Pi_{\mathsf{E}},q}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda),$$

where OM-CCA-A2L is defined in Figure 4.11.

The following technical lemma shows that an LOE scheme satisfies this property, assuming the hardness of the OMDL problem.

Lemma 3. *Let* Π_{E} *be an LOE scheme. Assuming the hardness of OMDL,* Π_{E} *is OM-CCA-A2L secure.*

Proof. We give a proof by reduction. Let $\mathcal{A}$ be a PPT adversary with non-

$\underline{\text{OM-CCA-A2L}^{\mathcal{A}}_{\Pi_{\mathsf{E}},q}}$

$Q := 0$

$(\mathsf{ek}, \mathsf{dk}) \leftarrow \Pi_{\mathsf{E}}.\mathsf{KGen}(1^\lambda)$

$r_1, \ldots, r_{q+1} \leftarrow\!\!\$\ \{0,1\}^\lambda$

$c_i \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}, r_i)$

$(r'_1, \ldots, r'_{q+1}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{A^2L}}_{\mathsf{dk},\Pi_{\mathsf{E}},\Pi_{\mathsf{ADP}}}}(\mathsf{ek}, (c_1, g^{r_1}), \ldots, (c_{q+1}, g^{r_{q+1}}))$

if $r'_i = r_i\ \forall i \in 1, \ldots, q+1 \wedge Q \leq q$ **then return** 1

else return 0

$\underline{\mathcal{O}^{\mathsf{A^2L}}_{\mathsf{dk},\Pi_{\mathsf{E}},\Pi_{\mathsf{ADP}}}(\mathsf{vk}, m, h, c, \tilde{\sigma})}$

check if $\mathsf{vk} \in \mathsf{Supp}(\Pi_{\mathsf{ADP}}.\mathsf{KGen}(1^\lambda))$

$\tilde{x} \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}, c)$

if $\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}, m, h, \tilde{\sigma}) = 1$ **and** $g^{\tilde{x}} = h$

  $Q := Q + 1$

  return $\sigma' \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}, \tilde{x})$

else return $\perp$

Figure 4.11: OM-CCA-A2L game

negligible advantage in the OM-CCA-A2L game. We now construct an adversary $\mathcal{R}$ which uses $\mathcal{A}$ to break the security of OMDL.

$\mathcal{R}$ is given $(h_1, \ldots, h_{q+1}) = (g^{r_1}, \ldots, g^{r_{q+1}})$ by the OMDL game. It will run $\mathcal{A}$ to attempt to obtain the $q+1$ discrete logarithms to win the game. Crucially, $\mathcal{R}$ must simulate $\mathcal{A}$'s oracle access to $\mathcal{O}^{\mathsf{A^2L}}_{\mathsf{sk},\Pi_{\mathsf{E}},\Pi_{\mathsf{ADP}}}$, which consists of at most q successful queries (but unlimited $\perp$ queries), while making at most q queries (of *any* kind) to its oracle $\mathsf{DL}(\cdot)$.

$\mathcal{R}$ proceeds as follows. First, it samples $q+1$ uniform λ-bit strings $(c^*_1, \ldots, c^*_{q+1})$. Note that these are identically distributed to outputs of $\mathcal{O}^{\mathsf{Enc}}$. It enters $(X_1, c^*_1), \ldots, (X_{q+1}, c^*_{q+1})$ into a table M, where the X_i are random variables. Now it sends $(c^*_1, h_1), \ldots, (c^*_{q+1}, h_{q+1})$ to the adversary $\mathcal{A}$.

Any queries $\mathcal{A}$ makes to the encryption scheme oracles ($\mathcal{O}^{\mathsf{KGen}}$, $\mathcal{O}^{\mathsf{Enc}}, \mathcal{O}^{\mathsf{Dec}}, \mathcal{O}^{\mathsf{Add}}$) and their corresponding responses are passed along unchanged by

1. If $c_i = c^*_j$ and $k_i = h_j$ for some j, it checks $\mathsf{PreVerify}(\mathsf{vk}_i, m_i, k_i, c_i) = 1$. If not, it returns $\perp$; otherwise, it queries $\mathsf{DL}(h_j)$ to get x_j and returns $\Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}_i, x_j)$ to $\mathcal{A}$.
2. If $c_i = c^*_j$ but $k_i \neq h_j$, $\mathcal{R}$ sends $\perp$ to $\mathcal{A}$.
3. If $(\cdot, c_i) \notin M$, $\mathcal{R}$ sends $\perp$ to $\mathcal{A}$.

4. Otherwise, let p_i be the plaintext entry corresponding to c_i in M. Notice that, by the linear-only property of the encryption scheme, p_i is a polynomial in $X_1, \ldots, X_{q+1}$ with $\deg(p_i) \leq 1$.

   (a) If $\deg(p_i) = 0$, p_i is some constant value x_j. In this case, $\mathcal{R}$ uses x_j to proceed as the normal $\mathcal{O}^{\mathsf{A^2L}}$ oracle does (checks if the pre-signature verifies and adapts it if so) and sends its output to $\mathcal{A}$.

   (b) If $\deg(p_i) = 1$, define $p_i := \alpha_0 + \alpha_1 X_1 + \ldots + \alpha_n X_{q+1}$. If $k_i = g^{\alpha_0} \prod_{k=1}^{q+1} h_k^{\alpha_k} = g^{p_i}$ and $\mathsf{PreVerify}(\mathsf{vk}_i, m_i, k_i, c_i) = 1$, $\mathcal{R}$ uses a query $\mathsf{DL}(k_i)$ to get x_j and outputs $\Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}_i, x_j)$. Otherwise, it sends $\perp$ to $\mathcal{A}$.

Observe that $\mathcal{R}$ returns $\perp$ *without querying* $\mathsf{DL}(\cdot)$ for all $\perp$ $\mathsf{A^2L}$-queries $\mathcal{A}$ makes. Thus it makes at most q queries to $\mathsf{DL}(\cdot)$. If $\mathcal{A}$ outputs winning values $(r_1, \ldots, r_{q+1})$, $\mathcal{R}$ outputs the same values, thereby winning the OMDL game. By assumption, $\mathcal{A}$ succeeds with non-negligible probability, and thus $\mathcal{R}$ also wins with non-negligible probability. This violates the OMDL assumption, implying that no such adversary $\mathcal{A}$ can exist. □

We are now ready to give the main theorem of this section.

Theorem 2. *Let Π_{E} be an LOE scheme, Π_{ADP} a secure adaptor signature scheme, and Π_{NIZK} a sound NIZK proof system. Assuming the hardness of OMDL, the A^2L^+ protocol is a secure blind conditional signature scheme.*

Proof. We argue about each property separately.

Lemma 4 (Blindness). *Assuming Π_{NIZK} is sound, the A^2L^+ scheme is blind in the LOE model.*

Proof. This holds information-theoretically. Fix any two $\mathsf{PPromise}$ executions. We now show, via a series of hybrid experiments, that the cases of $b = 0$ and $b = 1$ are statistically close.

Hybrid $\mathcal{H}_0$: Run $\mathsf{ExpBlnd}$ with $b = 0$.

Hybrid $\mathcal{H}_1$: In both runs of PSolve, sample $r \leftarrow\$ \mathcal{Z}_q$ and set $Y'' := g^r$ and $c'' \leftarrow \Pi_{\mathsf{E}}(\mathsf{pk}_H, r)$.

Hybrid $\mathcal{H}_2$: Compute c'' and Y'' honestly using τ_1 in the first run of PSolve and τ_0 in the second run of PSolve.

Hybrid $\mathcal{H}_3$: Run $\mathsf{ExpBlnd}$ with $b = 1$.

Claim 1. *For all PPT adversaries $\mathcal{A}$, $\mathcal{H}_0 \approx \mathcal{H}_1$.*

Proof. Y'' is g raised to a uniform element and c'' is an encryption of the same uniform element in both experiments, conditioned on the ciphertext provided by the Hub being well-formed. Thus, any distinguishing advantage necessarily corresponds to a violation of the soundness property of Π_{NIZK}. It follows that the executions are statistically indistinguishable. □

Claim 2. *For all PPT adversaries $\mathcal{A}$, $\mathcal{H}_1 \approx \mathcal{H}_2$.*

Proof. This holds by the same logic as Claim 1. $\square$

Claim 3. *For all PPT adversaries $\mathcal{A}$, $\mathcal{H}_2 \equiv \mathcal{H}_3$.*

Proof. The change is only syntactical and the executions are identical. $\square$

Hence, the cases of $b = 0$ and $b = 1$ are statistically indistinguishable. $\square$

Lemma 5 (Unlockability)**.** *Assuming that Π_{ADP} is witness extractable, pre-signature adaptable, and unforgeable the A^2L^+ scheme is unlockable.*

Proof. We consider two cases separately.

$(b_2 \wedge b_3) = 1$: First, let us consider the case in which $\mathcal{A}$ outputs a valid signature σ^A_{AH} while at the same time $s'' \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Ext}(\mathsf{vk}^A_{AH}, \tilde{\sigma}^A_{AH}, \sigma^A_{AH}, Y'')$ is not a valid witness for Y''. Then we can give a reduction which breaks witness extractability with non-negligible probability. The reduction samples a uniform element $r \leftarrow\$ \mathbb{Z}_q$ and runs $\mathcal{A}$. It sets $Y'' := g^r$ and uses the encryption key $\tilde{\mathsf{ek}}$ output by $\mathcal{A}$ compute $c'' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\tilde{\mathsf{ek}}, r)$. In the puzzle solver phase, it sends Y'', c'' and the witness extractability challenge $\tilde{\sigma}$ to $\mathcal{A}$ and outputs the signature σ it receives in response (note that this is perfectly indistinguishable from an honest run of the protocol). Then $\Pi_{\mathsf{ADP}}.\mathsf{Ext}(\mathsf{vk}^A_{AH}, \tilde{\sigma}, \sigma, Y'')$ is not a valid witness for Y'', but this violates the witness extractability of Π_{ADP}, and therefore the probability of this case occurring is negligible.

The above argument establishes that s'' is a valid witness for Y'' with all but negligible probability. Since $Y'' = Y \cdot g^{r+r'} = g^y \cdot g^{r+r'}$, the only valid witness for Y'' is $y + (r + r')$, and therefore $s'' = y + (r + r')$. Hence $y = s'' - (r + r')$ is a valid witness for the statement Y and thus also for Bob's pre-signature $\tilde{\sigma}^H_{HB}$ (recall that in the protocol, Bob explicitly checks the pre-signature validity of $\tilde{\sigma}_{HB}$ with respect to Y). By pre-signature adaptability of Π_{ADP}, we have that $\Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}^H_{HB}, m, \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}^H_{HB}, y)) = 1$ with probability 1. Therefore, the adversary succeeds in this case with negligible probability.

$(b_0 = 1) \vee (b_1 = 1)$: In this case, the adversary is able to produce a valid signature on a message without seeing any pre-signature on it. This only happens with negligible probability by the unforgeability of the adaptor signature scheme. $\square$

Lemma 6 (Unforgeability)**.** *Assuming the hardness of OMDL and that Π_{ADP} is witness extractable and unforgeable, the A^2L^+ scheme is unforgeable in the LOE model.*

Proof. We give a series of hybrid experiments, show they are indistinguishable, and prove by reduction to OM-CCA-A2L that no adversary exists with non-negligible advantage against the final hybrid.

Hybrid $\mathcal{H}_0$: This is the normal game $\mathsf{ExpUnforg}$ (Figure 4.10).

Hybrid $\mathcal{H}_1$: Simulate all NIZK proofs using $\Pi_{\mathsf{NIZK}}.\mathsf{Sim}$.

Hybrid $\mathcal{H}_2$: If $\exists\ i \in [q]$ such that $\mathsf{Verify}(\mathsf{vk}_i^H, m_i, \sigma_i) = 1$ and $(\mathsf{vk}_i^H, \cdot) \in \mathcal{L}$ but $(\mathsf{vk}_i^H, m_i) \notin \mathcal{L}$, return 0.

Hybrid $\mathcal{H}_3$: If $\exists\ i \in [q]$ such that $\mathsf{Verify}(\mathsf{vk}_i^H, m_i, \sigma_i) = 1$ and $g^{\mathsf{Ext}(\tilde{\sigma}_i, \sigma_i)} \neq Y_i$, return 0.

Claim 4. *For all PPT adversaries $\mathcal{A}$, $\mathcal{H}_0 \approx \mathcal{H}_1$.*

Proof. This follows directly from zero-knowledge of Π_{NIZK}. □

Claim 5. *For all PPT adversaries $\mathcal{A}$, $\mathcal{H}_1 \approx \mathcal{H}_2$.*

Proof. The hybrids differ only in the case where the attacker returns a valid signature on a message that was not part of the transcript. By the unforgeability of the adaptor signature, this happens only with negligible probability. □

Claim 6. *For all PPT adversaries $\mathcal{A}$, $\mathcal{H}_2 \approx \mathcal{H}_3$.*

Proof. Any distinguishing advantage corresponds to the case in which $\mathcal{A}$ outputs some tuple $(\mathsf{vk}_i^H, m_i, \sigma_i)$ such that, for corresponding $(Y_i, \tilde{\sigma}_i)$, $g^{\Pi_{\mathsf{ADP}}.\mathsf{Ext}(\tilde{\sigma}_i, \sigma_i)} \neq Y_i$. In this case, we can give a reduction to witness extractability of Π_{ADP}. The reduction runs the setup as in $\mathcal{H}_3$ and receives a verification key vk from the witness extractability game. It now picks some guess $i^* \leftarrow\$ \{1, \ldots q-1\}$ (where $q-1$ is the number of queries of the adversary) for the distinguishing index and starts $\mathcal{A}$ on $\tilde{\mathsf{ek}}$, behaving the same way as $\mathcal{H}_3$ for all oracle queries, except for the i^*-th interaction, in which it sets $\mathsf{vk}^H := \mathsf{vk}$. In the execution of $\mathsf{PPromise}$, it sends m_{i^*} to the witness extractability game and receives $\tilde{\sigma}$, which it gives to $\mathcal{A}$ instead of computing $\tilde{\sigma}_{HB}^H$ itself. Once $\mathcal{A}$ terminates and outputs $\{\mathsf{vk}_i^H, m_i, \sigma_i)\}_{i=1}^q$, the reduction sends σ_{i^*} to its game. If it guessed the distinguishing index i^* correctly, this is a winning signature. Suppose the distinguishing advantage is non-negligible. Since the guess is correct with probability $1/(q-1)$, the reduction violates witness extractability also with non-negligible advantage, which is a contradiction. Hence the two experiments must be computationally close. □

Now we give a reduction from hybrid $\mathcal{H}_3$ to OM-CCA-A2L. Suppose there exists an adversary $\mathcal{A}$ with non-negligible success probability in $\mathcal{H}_3$. We give a reduction that uses $\mathcal{A}$ to win the OM-CCA-A2L game. The reduction is given $(c_1, h_1), \ldots, (c_{q+1}, h_{q+1})$. It generates $(\tilde{\mathsf{ek}}, \tilde{\mathsf{dk}}) \leftarrow \Pi_{\mathsf{E}}.\mathsf{KGen}(1^\lambda)$ and $(\mathsf{vk}^H, \mathsf{sk}^H)$ as in $\mathcal{H}_3$ and starts $\mathcal{A}$ on input $\tilde{\mathsf{ek}}$. For $\mathcal{O}\mathsf{PPromise}$ queries, the reduction follows the same steps as $\mathcal{H}_3$ except it uses a different challenge h_i each time it generates a pre-signature. When $\mathcal{A}$ queries $\mathcal{O}\mathsf{PSolve}$, the reduction computes the completed signature σ_{AH}^A as the output of $\mathcal{O}^{\mathsf{A^2L}}$ run on $\mathcal{A}$'s inputs $(\mathsf{vk}_{AH}^A, m', Y'', c'', \sigma_{AH}^A)$. Note that since $\mathcal{A}$ makes at most q non-$\perp$ queries to $\mathcal{O}\mathsf{PSolve}$, the reduction also makes at most q non-$\perp$ queries to $\mathcal{O}^{\mathsf{A^2L}}$, as the oracles return $\perp$ in exactly the same cases.

Once $\mathcal{A}$ returns $q+1$ tuples $(\mathsf{vk}_j^H, m_j, \sigma_j)$, the reduction computes $r_i \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Ext}(\mathsf{vk}_j^H, \tilde{\sigma}_i, \sigma_j, h_i) \forall i, j \in [q+1]$ until it has $q+1$ non-$\perp$ values r_i (at most $(q+1)^2$ invocations of the algorithm) and outputs those values. Note that by the definition of $\mathcal{H}_3$, when $\mathcal{A}$ completes successfully, $g^{r_i} = h_i \ \forall i \in [q+1]$. By assumption, the reduction wins the OM-CCA-A2L game with non-negligible probability. This violates OM-CCA-A2L-security of Π_{E} (implied by Lemma 3), so no such adversary against $\mathcal{H}_3$ exists. Thus, no adversary with non-negligible success in ExpUnforg can exist either. □

The theorem follows directly from Lemmas 4 to 6. □

4.9 UC-Secure Blind Conditional Signatures

We now model security in the universal composability framework (Section 2.8) extended to support a global setup [CDPW07] in order to capture concurrent executions. We consider *static* corruptions, where the adversary announces at the beginning which parties it corrupts.

In our protocol, we assume the existence of a general-purpose UC-secure 2-party computation (2PC) protocol [HK07, CLOS02b], where two parties interact with the ideal functionality to compute a function $f(x, y)$ over their private inputs x and y.

4.9.1 Ideal Functionality

In Figure 4.12, we describe the ideal functionality $\mathcal{F}_{\mathsf{BCS}}$ that captures the functionality and security of BCS in the UC framework. The ideal functionality has three routines, namely for puzzle promise, puzzle solver, and open, which intuitively capture the functionality of BCS as discussed in Chapter 4. On a high level, $\mathcal{F}_{\mathsf{BCS}}$ captures *blindness* by sampling the puzzle identifiers pid and pid′, which correspond to puzzle promise and puzzle solve interactions, locally together, but never revealing them together to the hub. $\mathcal{F}_{\mathsf{BCS}}$ captures *atomicity* by returning a successful message (not aborting) for pid during open if and only if it sent a successful solved message during the puzzle solve interaction for the puzzle identifier pid′ (where pid and pid′ correspond to each other). Note that the above atomicity guarantee implies the game-based definitions of unlockability and unforgeability.

Our functionality $\mathcal{F}_{\mathsf{BCS}}$ is taken verbatim from the $\mathcal{F}_{\mathsf{A^2L}}$ functionality in [TMM21] except that we do not consider user registrations (as done in $\mathcal{F}_{\mathsf{A^2L}}$) to tackle *griefing attacks* [Rob19] in the coin mixing layer. These attacks are mounted by Bob starting many puzzle promise operations, each of which requires Hub to lock coins, whereas the corresponding puzzle solver interactions are never carried out. As a consequence, all of Hub's coins are locked and no longer available, which results in a form of denial of service attack. We argue that the issue does not concern the functionality or security of BCS as a cryptographic tool, but only affects the coin mixing protocol at the transaction layer. We emphasize that griefing attacks can be thwarted at this layer in both the formal model and the construction using the same ideas as in [TMM21].

Ideal Functionality $\mathcal{F}_{\mathsf{BCS}}$
Puzzle Promise: On input $(\mathsf{PPromise}, A)$ from B, $\mathcal{F}_{\mathsf{BCS}}$ proceeds as follows: • Send $(\mathsf{promise{-}req}, B)$ to H and $\mathcal{S}$. • Receive $(\mathsf{promise{-}res}, b)$ from H. • If $b = \bot$ then abort. • Sample $\mathsf{pid}, \mathsf{pid}' \leftarrow\$ \{0,1\}^\lambda$. • Store the tuple $(\mathsf{pid}, \mathsf{pid}', \bot)$ into $\mathcal{P}$. • Send $(\mathsf{promise}, (\mathsf{pid}, \mathsf{pid}'))$ to B, $(\mathsf{promise}, \mathsf{pid})$ to H, $(\mathsf{promise}, \mathsf{pid}')$ to A, and inform $\mathcal{S}$.
Puzzle Solver: On input $(\mathsf{PSolve}, B, \mathsf{pid}')$ from A, $\mathcal{F}_{\mathsf{BCS}}$ proceeds as follows: • If $(\cdot, \mathsf{pid}', \cdot) \notin \mathcal{P}$ then abort. • Send $(\mathsf{solve{-}req}, A, \mathsf{pid}')$ to H and $\mathcal{S}$. • Receive $(\mathsf{solve{-}res}, b)$ from H. • If $b = \bot$ then abort. • Update entry to $(\cdot, \mathsf{pid}', \top)$ in $\mathcal{P}$. • Send $(\mathsf{solved}, \mathsf{pid}', \top)$ to A, B and $\mathcal{S}$.
Open: On input $(\mathsf{Open}, \mathsf{pid})$ from B, $\mathcal{F}_{\mathsf{BCS}}$ proceeds as follows: • If $(\mathsf{pid}, \cdot, b) \notin \mathcal{P}$ or $b = \bot$ then send $(\mathsf{open}, \mathsf{pid}, \bot)$ to B and abort. Else send $(\mathsf{open}, \mathsf{pid}, \top)$ to B.

Figure 4.12: Ideal functionality $\mathcal{F}_{\mathsf{BCS}}$ (corresponds to $\mathcal{F}_{\mathsf{A^2L}}$ in [TMM21]). Portions related to griefing protection (i.e., registration) have been removed.

4.9.2 The $\mathrm{A^2L^{UC}}$ Protocol

We now describe our protocol $\mathrm{A^2L^{UC}}$ that realizes the ideal functionality $\mathcal{F}_{\mathsf{BCS}}$. We assume the following cryptographic building blocks:

- An adaptor signature scheme Π_{ADP} defined with respect to Π_{DS} and a hard relation R_{DL}.
- A UC-secure NIZK proof system Π_{NIZK} for the language

$$\mathcal{L} := \{(\mathsf{ek}, Y, c) : \exists s, \text{ s.t. } c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}, s) \wedge Y = g^s\}.$$

- A UC-secure 2PC protocol.
- A CCA-secure [GM82] encryption scheme $\Pi_{\mathsf{E}} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ with unique decryption keys.

The property of unique decryption keys is formalized below.

Definition 25 (Unique Decryption Keys)**.** *An encryption scheme* Π_{E} *has unique decryption keys if the* KGen *algorithm is of the following form:*

- *Sample* $\mathsf{dk} \leftarrow\!\!\$ \{0,1\}^{\lambda}$.
- *Run* $\mathsf{ek} \leftarrow \mathsf{KGen}(\mathsf{dk})$.

Furthermore, for all ek *output by* KGen*, there exists a unique* dk *such that* $\mathsf{ek} = \mathsf{KGen}(\mathsf{dk})$*. In other words,* KGen *is injective.*

This property is already satisfied by most natural public-key encryption schemes, but it can be generically achieved by augmenting the encryption key with a perfectly binding commitment $\mathsf{Com}(\mathsf{dk})$ to the decryption key dk.

Protocol Description We assume Alice and Hub have a key pair for the signature scheme Π_{DS}. Specifically, we have the verification-signing key pairs $(\mathsf{vk}^{H}_{HB}, \mathsf{sk}^{H}_{HB})$ and $(\mathsf{vk}^{A}_{AH}, \mathsf{sk}^{A}_{AH})$, belonging to Hub and Alice, respectively. We then have two messages $m := m_{HB}$ and $m' := m_{AH}$ for which the users wish to generate blind conditional signatures. The setup and open algorithms are formally described in Figure 4.13. The puzzle promise and puzzle solver of $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$ are formally described in Figure 4.14 and Figure 4.15, respectively. For ease of understanding, we briefly describe below our $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$ protocol in terms of the differences with the $\mathrm{A}^2\mathrm{L}$ protocol (Figures 4.5 and 4.6).

- The setup algorithm (Figure 4.13) of $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$ generates the keys of Hub, which are the keys for the (CCA-secure) encryption scheme Π_{E}.
- In $\mathsf{PPromise}$ of $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$ (Figure 4.14),
  - The NIZK proof system is UC-secure.
  - Bob no longer re-randomizes the instance or the ciphertext. Therefore, we drop the re-randomization steps (line 9 and 10) of $\mathsf{PPromise}$ in $\mathrm{A}^2\mathrm{L}$ (Figure 4.5). Simply set the puzzle to $\tau := (m_{HB}, \tilde{\sigma}^{H}_{HB}, (Y, c))$.
- In PSolve of $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$ (Figure 4.15),
  - Alice no longer sends the ciphertext to Hub (line 5 of Figure 4.6). We therefore remove the local decryption step (line 6 of Figure 4.6), and replace it with a 2PC protocol (line 6 of Figure 4.15).
  - At the end of the 2PC protocol, Alice receives $\perp$, while Hub receives the value z. Hub additionally checks if $Y' = g^z$ (line 7) and uses z to adapt the pre-signature $\tilde{\sigma}^{A}_{AH}$ to signature σ^{A}_{AH}.
  - We add a check for Alice (line 10) that σ^{A}_{AH} is a valid signature before extracting the witness z' in line 12.
- The Open algorithm (Figure 4.13) is the same as in Figure 4.7 of $\mathrm{A}^2\mathrm{L}$, except we skip removing the randomness factor. The algorithm in Figure 4.13 now simply adapts a pre-signature $\tilde{\sigma}$ to a valid signature σ which it returns as output.

$\mathsf{Setup}(1^\lambda)$	$\mathsf{Open}(\tau, s)$
$(\mathsf{ek}_H, \mathsf{dk}_H) \leftarrow \Pi_{\mathsf{E}}.\mathsf{KGen}(1^\lambda)$	**parse** $\tau := (\cdot, \tilde{\sigma}, \cdot)$
set $\tilde{\mathsf{pk}} := \mathsf{ek}_H, \tilde{\mathsf{sk}} := \mathsf{dk}_H$	$\sigma \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}, s)$
return $(\tilde{\mathsf{pk}}, \tilde{\mathsf{sk}})$	**return** σ

Figure 4.13: Setup and Open algorithms of our conditional puzzle construction

Public parameters: group description $(\mathbb{G}, g, q)$, message m_{HB}

$\mathsf{PPromise}\langle H(\mathsf{dk}_H, \mathsf{sk}_{HB}^H), \cdot\rangle$		$\mathsf{PPromise}\langle \cdot, B(\mathsf{ek}_H, \mathsf{vk}_{HB}^H)\rangle$
$s \leftarrow\!\!\$\; \mathbb{Z}_p, Y := g^s$		
$c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, s)$		
$\pi_s \leftarrow \mathsf{NIZK}.\mathsf{Prove}((\mathsf{ek}_H, Y, c), s)$		
$\tilde{\sigma}_{HB}^H \leftarrow \Pi_{\mathsf{ADP}}.\tilde{\sigma}(\mathsf{sk}_{HB}^H, m_{HB}, Y)$		
	$\xrightarrow{Y, c, \pi_s, \tilde{\sigma}_{HB}^H}$	
		If $\mathsf{NIZK}.\mathsf{Verify}((\mathsf{ek}_H, Y, c), \pi_s) \neq 1$ **then return** $\perp$
		If $\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}_{HB}^H, m_{HB}, Y, \tilde{\sigma}_{HB}^H) \neq 1$ **then**
		return $\perp$
		set $\tau := (m_{HB}, \tilde{\sigma}_{HB}^H, (Y, c))$
return $\perp$		**return** τ

Figure 4.14: Puzzle promise protocol of $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$

4.9.3 Security Analysis

We now show that $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$ satisfies UC-security. In favor of a simpler analysis, we assume that the verification keys of all parties are honestly generated. In practice, this can be enforced by augmenting keys with NIZKs that certify their validity [Bol03, LOS+06].

Theorem 3. *Let Π_{E} be a CCA-secure encryption scheme, Π_{ADP} a secure adaptor signature scheme,* 2PC *a UC-secure two-party computation protocol, and Π_{NIZK} a UC-secure NIZK for the language $\mathcal{L}$ above. Then the A^2L^{UC} protocol UC-realizes $\mathcal{F}_{\mathsf{BCS}}$.*

Proof. We proceed by describing the UC simulator and arguing about indistinguishability from the real execution of the protocol. We consider the cases where the adversary corrupts a different subset of parties separately. We describe the simulator for a single session and the security of the overall interaction is established via a standard hybrid argument.

H Corrupted We first give a simulator $\mathcal{S}_H$, then give a series of hybrid experiments that gradually change the real experiment (i.e., the construction in Figures 4.14 and 4.15) into the ideal experiment given by the interaction of the corrupted H and the simulator $\mathcal{S}_H$, which has access to $\mathcal{F}_{\mathsf{BCS}}$.

Public parameters: group description $(\mathbb{G}, g, q)$, message m_{AH}

$\mathsf{PSolve}\langle A(\mathsf{sk}^A_{AH}, \mathsf{ek}_H, \tau), \cdot\rangle$		$\mathsf{PSolve}\langle \cdot, H(\mathsf{dk}_H, \mathsf{vk}^A_{AH})\rangle$
parse $\tau := (\cdot, \cdot, (Y, c))$		
$r \leftarrow\!\$\, \mathbb{Z}_p, Y' := Y \cdot g^r$		
$\tilde{\sigma}^A_{AH} \leftarrow \Pi_{\mathsf{ADP}}.\tilde{\sigma}(\mathsf{sk}^A_{AH}, m_{AH}, Y')$		
	$\xrightarrow{Y', \tilde{\sigma}^A_{AH}}$	
		If $\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}^A_{AH}, m_{AH}, Y', \tilde{\sigma}^A_{AH}) \neq 1$ **then** **return** $\perp$
	$\mathsf{2PC}((r, c), (\mathsf{dk}_H))$: **if** $\mathsf{ek}_H \neq \Pi_{\mathsf{E}}.\mathsf{Gen}(\mathsf{dk}_H)$ **then** abort $s^* \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c)$ $z := s^* + r$ **return** $((\perp), (z))$	
		If $Y' \neq g^z$ **then return** $\perp$
		$\sigma^A_{AH} \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}^A_{AH}, z)$
	$\xleftarrow{\sigma^A_{AH}}$	
If $\Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}^A_{AH}, m_{AH}, \sigma^A_{AH}) \neq 1$ **then**		
return $\perp$		
$z' \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Ext}(\tilde{\sigma}^A_{AH}, \sigma^A_{AH}, Y')$		
$s := z' - r$		
return (σ^A_{AH}, s)		**return** σ^A_{AH}

Figure 4.15: Puzzle solver protocol of $\mathrm{A}^2\mathrm{L}^{\mathrm{UC}}$

Simulator $\mathcal{S}_H$: Upon receiving $(\mathsf{promise{-}req}, B)$ from $\mathcal{F}_{\mathsf{BCS}}$, $\mathcal{S}_H$ proceeds as follows:

1. Ask the attacker to initiate a session and receive in return $(Y, c, \pi_s, \tilde{\sigma}^H_{HB})$. If $\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}^H_{HB}, m_{HB}, Y, \tilde{\sigma}^H_{HB}) = 1$ and $\mathsf{NIZK.V}((\mathsf{ek}_H, Y, c), \pi_s) = 1$, proceed as in the protocol and send $(\mathsf{promise{-}res}, \top)$ to $\mathcal{F}_{\mathsf{BCS}}$. Otherwise, abort and send $(\mathsf{promise{-}res}, \perp)$.

2. Receive $(\mathsf{promise}, \mathsf{pid})$ from $\mathcal{F}_{\mathsf{BCS}}$.

3. Upon receiving $(\mathsf{solve{-}req}, A, \mathsf{pid}')$ from $\mathcal{F}_{\mathsf{BCS}}$ at some later point, sample $y' \leftarrow\!\$\, \mathbb{Z}_q$ and generate keys $(\mathsf{vk}^A_{AH}, \mathsf{sk}^A_{AH}) \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{KGen}(1^\lambda)$. Compute $Y' \leftarrow g^{y'}$, $\tilde{\sigma}^A_{AH} \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{PreSign}(\mathsf{sk}^A_{AH}, m_{AH}, Y')$ and send them to the attacker.

4. When the attacker initiates the 2PC, run the 2PC simulator to recover its input dk_H. If $\mathsf{ek}_H \neq \Pi_{\mathsf{E}}.\mathsf{KGen}(\mathsf{dk}_H)$, program the output of the 2PC to $\perp$, otherwise to y'.

5. Receive σ^A_{AH} in response from the attacker and check if $\Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}^A_{AH}, m_{AH}, \sigma^A_{AH}) = 1$. Additionally check if $\Pi_{\mathsf{ADP}}.\mathsf{Ext}(\tilde{\sigma}^A_{AH}, \sigma^A_{AH}, Y') = y'$. If both checks pass, send $(\mathsf{solve{-}res}, \top)$ to $\mathcal{F}_{\mathsf{BCS}}$, compute $s \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c)$, and output (σ^A_{AH}, s) as in the protocol; otherwise, send $(\mathsf{solve{-}res}, \bot)$ and abort.

6. If, at any point before the successful completion of step 4, the attacker produces a valid signature σ^A_{AH}, or at any point in the protocol (including after step 4), a valid signature on a message $m'_{AH} \neq m_{AH}$, send $(\mathsf{solve{-}res}, \bot)$ to $\mathcal{F}_{\mathsf{BCS}}$ and abort.

Hybrid $\mathcal{H}_0$: This corresponds to the real protocol (Figures 4.14 and 4.15).

Hybrid $\mathcal{H}_1$: Simulate the 2PC (Fig. 4.15, line 6) and send the output z to H.

Hybrid $\mathcal{H}_2$: Replace Y' with $Y'' := g^{y'}$ where $y' \leftarrow\!\!\$\ \mathbb{Z}_q$ (Fig. 4.15, line 2). If $\Pi_{\mathsf{E}}.\mathsf{KGen}(\mathsf{dk}_H) = \mathsf{ek}_H$, send y' to H instead of z; otherwise, send $\bot$.

Hybrid $\mathcal{H}_3$: Abort if $z' \neq y'$ (after line 12 of Fig. 4.15).

Hybrid $\mathcal{H}_4$: Abort if any valid signature σ^A_{AH} is received on a different message $m'_{AH} \neq m_{AH}$ or on any message before the 2PC has successfully completed.

Claim 7. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}$$

Proof. This follows directly from the security of the 2PC protocol. □

Claim 8. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}$$

Proof. By the uniqueness of the decryption key and correctness of Π_{E}, $\mathsf{ek}_H = \Pi_{\mathsf{E}}.\mathsf{KGen}(\mathsf{dk}_H)$ implies $\Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, \Pi_{\mathsf{Enc}}.\mathsf{Enc}(\mathsf{ek}_H, m)) = m$ for all m in the message space. Thus, the output of the 2PC z is necessarily $s+r$, where $s \in \mathbb{Z}_q$ such that $c = \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, s) \ \wedge\ Y = g^s$ (this is guaranteed by the NIZK). Since r is uniformly random, y' is identically distributed to $z = s+r$. The same holds for Y'' and $Y' = Y \cdot g^r$. Furthermore, it still holds that y' is the discrete logarithm of Y'' (cf. z and Y'). □

Claim 9. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}$$

Proof. If $z' \neq y'$, by the uniqueness of dlog witnesses $g^{z'} \neq Y''$. By the witness extractability of Π_{ADP}, $\Pr[g^{z'} \neq Y'' \ \wedge\ \Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}^A_{AH}, m_{AH}, \sigma^A_{AH}) = 1]$ is negligible, so the abort only happens with negligible probability. □

Claim 10. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}$$

Proof. Any distinguishing advantage implies a case in which $\mathcal{A}$ outputs some valid signature σ_{AH}^A for some message m'_{AH} for which it has potentially been given a presignature $\tilde{\sigma}_{AH}^A$ and corresponding statement Y. This signature is a winning instance in the unforgeability experiment for Π_{ADP}, but by assumption this only occurs with negligible probability, and so the distinguishing advantage must be negligible. Therefore the experiments are statistically close. □

Claim 11. *For all PPT distinguishers $\mathcal{E}$,*

$$\text{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \equiv \text{EXEC}_{\mathcal{F}_{\mathsf{BCS}},\mathcal{S},\mathcal{E}}$$

Proof. $\mathcal{H}_4$ is identical to the ideal world. □

A,B Corrupted Again, we give a simulator $\mathcal{S}_{AB}$ that interacts with $\mathcal{F}_{\mathsf{BCS}}$ and show by a series of hybrids that our protocol is indistinguishable from ideal experiment in which the corrupted parties interact with the simulator $\mathcal{S}_{AB}$.

Simulator $\mathcal{S}_{AB}$: When a recipient Bob indicates he would like to initiate a transaction, $\mathcal{S}_{AB}$ proceeds as follows:

1. Send $(\mathsf{PPromise}, A)$ to $\mathcal{F}_{\mathsf{BCS}}$.

2. Upon receiving $(\mathsf{promise}, (\mathsf{pid}, \mathsf{pid}'))$ from $\mathcal{F}_{\mathsf{BCS}}$, sample a uniform value $s \leftarrow\$ \mathbb{Z}_q$ and compute $Y \leftarrow g^s$. Generate keys $(\mathsf{ek}_H, \mathsf{dk}_H) \leftarrow \Pi_{\mathsf{E}}.\mathsf{KGen}(1^\lambda)$ and $(\mathsf{vk}_{HB}^H, \mathsf{sk}_{HB}^H) \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{KGen}(1^\lambda)$; let $c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, 0)$ and $\tilde{\sigma}_{HB}^H \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{PreSign}(\mathsf{sk}_{HB}^H, m_{HB}, Y)$. Simulate the NIZK $\pi_s \leftarrow \mathsf{NIZK}.\mathsf{Sim}(\mathsf{td}, (\mathsf{ek}_H, Y, c))$. Finally, pre-compute $\sigma_{HB}^H \leftarrow \Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}_{HB}^H, s)$ and save $((\mathsf{pid}, \mathsf{pid}'), (Y, c, s, \sigma_{HB}^H), \perp)$ into a table $\mathcal{P}$. Send $(Y, c, \pi_s, \tilde{\sigma}_{HB}^H)$ to the attacker (who is impersonating Bob).

3. At a later point in time, the attacker sends $(Y', \tilde{\sigma}_{AH}^A)$ on behalf of Alice. If $\Pi_{\mathsf{ADP}}.\mathsf{PreVerify}(\mathsf{vk}_{AH}^A, m_{AH}, Y', \tilde{\sigma}_{AH}^A) \neq 1$, abort.

4. When the attacker initiates the 2PC, run the 2PC simulator to recover its inputs (c^*, r^*); compute the result $(\perp)$ and return it to the attacker.

5. Depending on whether or not $c^* \in \mathcal{P}$ do the following:

   (a) If $c^* \in \mathcal{P}$, retrieve the corresponding Y, s, and pid'. Check that $Y' = Y \cdot g^{r^*}$ (if not, abort); send $\Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}_{AH}^A, s + r^*)$ to the attacker masquerading as Alice and $(\mathsf{PSolve}, B, \mathsf{pid}')$ to $\mathcal{F}_{\mathsf{BCS}}$. Update the last element of the entry in $\mathcal{P}$ to $\top$.

   (b) If $c^* \notin \mathcal{P}$, compute $z' \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c^*) + r^*$. Check that $Y' = g^{z'}$ (if not, abort) and send $\Pi_{\mathsf{ADP}}.\mathsf{Adapt}(\tilde{\sigma}_{AH}^A, z')$ to the attacker. Send nothing to $\mathcal{F}_{\mathsf{BCS}}$. (Note that this corresponds to the case where some party Alice is paying Hub without Bob initiating the interaction, which is something that she can do at any time.)

6. When the attacker outputs some valid signature σ^H_{HB}, check that the following conditions hold: $\Pi_{\mathsf{ADP}}.\mathsf{Verify}(\mathsf{vk}^H_{HB}, m_{HB}, \sigma^H_{HB}) = 1$ and $((\mathsf{pid}, \cdot), (\cdot, \cdot, \cdot, \sigma^H_{HB}), \top) \in \mathcal{P}$. If so, send $(\mathsf{Open}, \mathsf{pid})$ to $\mathcal{F}_{\mathsf{BCS}}$; otherwise, abort.

Hybrid $\mathcal{H}_0$: This corresponds to the real protocol (Figures 4.14 and 4.15).

Hybrid $\mathcal{H}_1$: Replace the honestly-computed NIZK π_s (Figure 4.14, line 4) with a simulated proof.

Hybrid $\mathcal{H}_2$: Simulate the 2PC (Figure 4.15, line 6).

Hybrid $\mathcal{H}_3$: Add the list $\mathcal{P}$ and step 5 of the simulator (in particular, case 5a) to Figure 4.15, line 7-10.

Hybrid $\mathcal{H}_4$: Replace c (Figure 4.14, line 2) with an encryption of zero.

Hybrid $\mathcal{H}_5$: When Bob outputs a valid signature, abort if $(\cdot, (\cdot, \cdot, \cdot, \sigma^H_{HB}), b) \in \mathcal{P}$ and $b \neq \top$.

Claim 12. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}$$

Proof. This follows directly from the zero-knowledge property of the NIZK. □

Claim 13. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}$$

Proof. This follows directly from the UC-security of the 2PC protocol. □

Claim 14. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \equiv \mathrm{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}$$

Proof. By definition, for $c^* \in \mathcal{P}$, the corresponding s, Y in $\mathcal{P}$ are $\Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c^*)$ and g^s, respectively. Therefore $z' = s + r^*$ and the case of $c^* \in \mathcal{P}$ is handled in the same way as all cases were in the previous hybrid experiment. □

Claim 15. *For all PPT distinguishers $\mathcal{E}$,*

$$\mathrm{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}$$

Proof. Suppose towards a contradiction that $\mathcal{E}$ can distinguish the two executions with nonnegligible probability. We give a reduction to the CCA-security game of Π_{E}. The reduction sets $m_0 := s$ and $m_1 := 0$, sends them to the CCA game, and receives c. It then acts as hub in its interaction with $\mathcal{E}$, computing everything as in Hybrid 3, except for c, which it sets to the ciphertext it received from the game. When it needs to decrypt c^* it uses the CCA decryption oracle. At the end of the execution, based on $\mathcal{E}$'s guess, it outputs a bit to the CCA game (0 if $\mathcal{E}$ guesses $\mathcal{H}_3$, 1 otherwise), which will be correct with nonnegligible advantage. This violates the CCA-security of Π_{E}, so the two executions must be indistinguishable. □

Claim 16. *For all PPT distinguishers $\mathcal{E}$,*

$$\text{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \approx \text{EXEC}_{\mathcal{H}_5,\mathcal{A},\mathcal{E}}$$

Proof. If $b \neq \top$, Alice did not receive the completed signature σ^A_{AH} for that session and thus cannot recover the secret s to send to Bob. This means Bob's signature σ^H_{HB} was created without knowing the witness for the pre-signature $\tilde{\sigma}^H_{HB}$, which, by aEUF-CMA of Π_{ADP}, can only happen with negligible probability. Thus the abort also only happens with negligible probability and the two experiments are indistinguishable. □

Claim 17. *For all PPT distinguishers $\mathcal{E}$,*

$$\text{EXEC}_{\mathcal{H}_5,\mathcal{A},\mathcal{E}} \equiv \text{EXEC}_{\mathcal{F}_{\mathsf{BCS}},\mathcal{S},\mathcal{E}}$$

Proof. $\mathcal{H}_5$ is identical to the ideal world. □

A,H Corrupted This case is trivial, as B has no secret information and the simulator therefore simply follows the protocol.

H,B Corrupted The simulator in this case follows the protocol honestly. If hub publishes a valid signature σ^A_{AH} on a transaction m that is not in the simulator's (acting as Alice) transcript, the simulator aborts. This means that the adversary was able to forge a signature σ^A_{AH} on some transaction m for which it did not previously receive a pre-signature $\tilde{\sigma}^A_{AH}$. By EUF-CMA of the adaptor signature scheme, this case occurs with negligible probability and thus for all PPT distinguishers $\mathcal{E}$, the real world (an honest execution of the protocol) and the ideal world (an interaction with the simulator) are indistinguishable. □

4.10 Performance Evaluation

4.10.1 $\mathbf{A^2L^+}$

Recall that we use an encryption scheme Π_{E} in the LOE model. Below we present an instantiation of such a Π_{E}.

Instantiating Linear-Only Encryption As shown in [BCI$^+$13] it is not sufficient to instantiate this with any linearly homomorphic encryption (e.g., ElGamal). Though the scheme may not support homomorphic operations beyond linear, it may still have *obliviously sampleable ciphertexts*, i.e., the ability to sample a ciphertext without knowing the underlying plaintext. Note that this falls outside the LOE model, since there is no oracle that implements this functionality. Thus, as suggested in [BCI$^+$13] we implement an additional safeguard needed to prevent oblivious sampling. Given a linearly homomorphic encryption scheme $\Pi^*_{\mathsf{E}} := (\mathsf{KGen}^*, \mathsf{Enc}^*, \mathsf{Dec}^*)$ over $\mathbb{Z}_p$, we define a candidate LOE $\Pi_{\mathsf{E}} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ as follows:

		Class group				Prime order group $\mathbb{G}$		mod q		#H
		Exp	Op	Inv	DLog	Exp	Op	$\times$	$+$	
A^{2}L (insecure)	Schnorr	18	12	1	1	13	8	4	9	6
	ECDSA	18	12	1	1	27	8	17	10	11
A^{2}L$^+$	Schnorr	28	20	2	2	14	9	5	9	6
	ECDSA	28	20	2	2	32	10	21	12	11

Table 4.1: Operations in A^{2}L and A^{2}L$^+$ when instantiated with Schnorr or ECDSA adaptor signatures [AEE$^+$21b]. We give the number group exponentiations (Exp) and group operations (Op) in both class groups and groups $\mathbb{G}$ of prime order p, where $\log p = n$. Group element inversions (Inv) only occur in class groups. Decryption of a class group ciphertext also involves solving a discrete logarithm in a class group (DLog). We denote by #H the number of hash computations.

- $\mathsf{KGen}(1^\lambda)$: Sample $(\mathsf{ek}^*, \mathsf{dk}^*) \leftarrow \mathsf{KGen}^*(1^\lambda)$ and some $\alpha \leftarrow\$\ \mathbb{Z}_p$. Return $\mathsf{dk} := (\mathsf{dk}^*, \alpha)$ as the decryption key and $\mathsf{ek} := (\mathsf{ek}^*, \mathsf{Enc}^*(\mathsf{ek}^*, \alpha))$ as the encryption key.
- $\mathsf{Enc}(\mathsf{ek}^*, x)$: Compute c as $(\mathsf{Enc}^*(\mathsf{ek}^*, x), \mathsf{Enc}^*(\mathsf{ek}^*, \alpha \cdot x))$, where $\mathsf{Enc}^*(\mathsf{ek}^*, \alpha \cdot x)$ is computed homomorphically using ek.
- $\mathsf{Dec}(\mathsf{dk}^*, c)$: Parse c as (c_0, c_1) and compute $x_0 \leftarrow \mathsf{Dec}^*(\mathsf{dk}^*, c_0)$ and $x_1 \leftarrow \mathsf{Dec}^*(\mathsf{dk}^*, c_1)$. If $x_1 = \alpha \cdot x_0$ return x_0, else return $\perp$.

We note that the security of Π_{E} follows from the security of Π^*_{E}. Intuitively, we prevent oblivious ciphertext sampling, since it is infeasible for an adversary to sample a ciphertext component c_0 that is consistent with c_1 without knowing the plaintext of c_0.

Added Costs The new consistency check by the hub in PSolve adds 1 group operation and group exponentiation (Schnorr) or 5 group operations and 2 group exponentiations (ECDSA). The check on Alice's verification key vk^A_{AH} adds 3 modular multiplications and 2 modular additions in the ECDSA case. Furthermore, applying the LOE transformation described above to the CL encryption scheme results in a doubled ciphertext size and a corresponding increase in the operation count for decryption. We summarize the costs of A^{2}L and A^{2}L$^+$ in Section 4.10.1.

4.10.2 A^{2}L$^{\mathbf{UC}}$

Compared to A^{2}L$^+$, our A^{2}L$^{\mathrm{UC}}$ protocol removes the check on vk^A_{AH}, adds a signature verification, and moves the re-randomization and decryption into the 2PC. Additionally, Π_{E} is now required to be CCA-secure and the NIZK used must be UC-secure. The cost of the first two changes is minimal (net 1 group exponentiation, 1 group operation, and 1 hash computation); the most significant overhead is the result of the 2PC computation and the NIZK.

Assuming the CCA-secure Π_{E} in the 2PC is instantiated with the (prime-order-based) Cramer-Shoup cryptosystem [CS98] with SHA3-256 [Nat15] as the hash function, this incurs an overhead of 11 exponentiations, 9 multiplications, and 1 division in a group of prime order p and $\lceil \frac{3\lambda}{1088} \rceil \cdot 38400$ binary (AND) operations, where the security parameter λ equals $\log p$. Because the 2PC requires a mix of arithmetic and binary operations, a mixed-circuit 2PC protocol as implemented e.g. in [Kel20] could be used. Additionally, UC security of the NIZK can be achieved by replacing the use of the Fiat-Shamir transform in $\mathrm{A^2L}$ (and $\mathrm{A^2L^+}$) with the Fischlin transform, incurring a cost of roughly $\mathcal{O}(\log(\lambda))$ parallel repetitions of the base Fiat-Shamir NIZK. We stress that we view $\mathrm{A^2L^{UC}}$ as a proof-of-concept protocol showing the feasibility of achieving UC-secure blind conditional signatures and leave the problem of constructing an efficient UC-secure realization as an interesting direction for future work.

Chapter 5

Fair and Non-interactive On-chain Voting and Auctions[‡]

Contents

Auctions and voting are essential applications of Web3. For example, decentralized marketplaces run auctions to sell digital goods like non-fungible

[‡] *Portions of this section have been adapted from [GSZB24].*

tokens (NFTs) [Ope23] or domain names [XWY+21], while decentralized autonomous organizations (DAOs) deploy voting schemes to enact decentralized governance [Opt23b, FFH+24]. Most auction or voting schemes currently deployed on blockchains, e.g., OpenSea NFT auctions or Uniswap governance [FMW22], do not hide user bids/votes during the submission phase. This can negatively influence user behavior, for example, by vote herding or discouraging participation [EL04, GY19, SY03]. It can also cause network congestion as users engage in bidding wars, leading to surges in transaction fees and causing a negative externality for the entire network.

Privacy as a means for fairness These negative outcomes can be avoided if a protocol offers fairness, i.e., leaks no information about votes or bids until the end of the submission phase. A natural way to achieve this is to hide everything but the outcome. Existing works keep submissions hidden by introducing one or more trusted authorities who are still able to view all submissions and are trusted to correctly compute the result [BCD+09, AOZZ15] or prove its correctness in zero-knowledge (ZK) [Adi08, Pri23, GG22, Pri]. Others rely on advanced cryptographic primitives [CGGI16, dLNS17], e.g., (fully) homomorphic encryption ((F)HE [Gen09]). Neither of these approaches is suitable for an on-chain setting. Relying on a trusted third party (or a threshold of them) is at odds with the ethos of decentralization, whereas advanced cryptographic primitives are impractical on-chain today, incurring high computation or storage fees.

In any case, the privacy guarantees offered by these protocols are stronger than is necessary for achieving fairness: bids/ballots need only remain hidden *until the end of the submission phase*. This is acceptable in practice, with some deployed systems (e.g., ENS domain name bids [XWY+21]) achieving fairness via this weaker notion of privacy. We refer to this weaker notion as *tally-privacy* (as opposed to everlasting privacy). In fact, limited privacy can actually be a desideratum in certain settings, e.g., in representative democracies or in DAOs where delegates' votes are published to encourage accountability.

A common paradigm for tally-private, trustless protocols is a two-round commit-reveal protocol [FOO93, GY19, TAF+23]. In the first round, every party commits to their bid or ballot. In the second round, they open their commitments, and the winner is determined. Due to the lightweight cryptography used, these schemes are efficient enough to be deployed on-chain. However, their interactivity is a usability hurdle that causes friction in the protocols' execution since users must remember to come online and reveal their submissions after the voting or bidding period has elapsed. The reveal phase can also be an avenue for censorship: an attacker can bribe block proposers to exclude the openings of certain bids or ballots until after the result has been determined [PRF23].

We summarize these approaches for private voting and auctions in Table 5.1.

Approach	No TTPs	Non-interactive voting	Everlasting privacy	Practical
MPC [BDJ+06, AOZZ15]	○	●	●	●
ZK proofs [Adi08, Pri23, GG22, Pri]	○	●	●	●
FHE [CGGI16, dLNS17]	○	●	●	○
HE+ZK proofs [dLNS17]	○	●	●	◐
Commit-reveal [FOO93, GY19]	●	○	○	●
TLPs+HE [CJSS21] (Section 5.7.2)	◐	●	○	●
Cicada (this work)	●*	●	◐	●

Table 5.1: Major approaches for tally-private auction/voting schemes. MPC stands for secure multi-party computation. [dLNS17] aims to be practical but uses lattice-based cryptography, which is not feasible on-chain today. [AOZZ15, CJSS21] require a trusted setup but no TTP. The asterisk indicates that our scheme does not inherently require any TTPs (in particular, if the class group HTLP construction (Section 5.6.1) is used, Cicada has a transparent setup). Everlasting privacy can be added via an extension (Section 5.7.1).

5.1 Related Work

5.1.1 Voting and Auctions

The cryptographic literature on voting schemes [HMMP23] and sealed-bid auctions [FR96, HTK98, Cac99, NPS99, BDJ+06, BCD+09, PRST06] is enormous, dating to the 1990s. Most of these schemes are unsuitable to a fully decentralized and trust-minimized setting due to their inefficiency and/or reliance on trusted parties, e.g., tally authorities, servers running the public bulletin board, or auctioneers. Here, we review auction and voting protocols specifically designed to 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. [MSH17] proposed a “boardroom” voting protocol for DAO governance. Their protocol’s main disadvantage is that it can be aborted by a single party. Groth [Gro05] and Boneh et al. [BBC+23] 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 [MT19]. 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 [GY18], albeit with a *trusted auctioneer*. Bag et al. [BHSR19] 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 [HZ06]. Tyagi et al. proposed Riggs [TAF$^+$23], a fair non-interactive auction scheme using timed commitments [FKPS21, §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, Riggs is not scalable to a real-world setting with thousands of users. To achieve practicality, Riggs relies on an optimistic second round in which users voluntarily open their puzzles. Our approach is more practical as one only needs to solve a single (or handful of) TLPs, even in the worst case. Chvojka et al. suggest a TLP-based protocol for both e-voting and auctions [CJSS21], but their protocol has a per-election/auction trusted setup. We observe that HTLPs can help reduce (but not fully eliminate) the trust assumption by enabling a distributed per-protocol setup. We describe this idea, which may be of independent interest, in Section 5.7.2.

5.1.2 Time-based Cryptography

Time-based cryptography, which uses inherently sequential functions to delay the revelation of information, also has a lengthy history dating to the introduction of time-lock encryption in 1996 [RSW96]. Numerous variants have emerged since then, including timed commitments [BN00], proofs-of-sequential-work [MMV13], VDFs [BBBF18], and HTLPs [MT19], which we employ here. For a recent survey, we refer the reader to Medley et al. [MLQ23]. The only practical work we know of taking advantage of HTLPs is Bicorn [CATB23], which builds a distributed randomness beacon with a single aggregate HTLP for an arbitrary number of entropy contributors.

Delay encryption [BD21] constitutes the most recent development in time-based cryptography. It allows time-lock encryption of a message to an identity by combining identity-based encryption with inherently sequential computation. However, the only construction is based on isogenies and has enormous memory requirements ($\approx$ 12 TiB), making it impractical.

One can emulate time-based cryptography by applying stronger assumptions than the sequentiality of repeated (modular) squaring in groups of unknown order. McFly [DHMW23] and tlock [GMR23] build time-lock encryption from threshold trust, i.e., by assuming that a subset of signers reliably releases a threshold BLS signature [BLS01] on the current timestamp. Even though this strong assumption is at odds with our setting, we view this as a promising alternate research direction for building efficient and fair on-chain protocols. In this approach, submissions would be encrypted to a timestamp and could only be decrypted with the knowledge of a BLS signature released at the particular timestamp. This still requires linear work and storage in the number of submissions, so enabling aggregation of multiple time-lock ciphertexts is necessary to make this approach suitable for voting and auction applications. Aggregation is not currently possible in the aforementioned schemes as they lack any homomorphism, and we leave this question to future work.

5.2 Our Contributions

This work introduces Cicada, a framework for fair and non-interactive auctions and voting.[1] Cicada uses time-lock puzzles (TLPs) [RSW96] to achieve tally-privacy in a non-interactive, trustless, and efficient way. 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, we use *homomorphic* TLPs (HTLPs) to combine bids/ballots on-the-fly into a smaller number of TLPs which is independent of the number of users. Therefore, besides removing interactivity, Cicada offers improved storage compared to commit-reveal schemes, which incur linear storage costs. Since fully homomorphic TLPs are not practically efficient, we only require additive HTLPs, which have efficient constructions [MT19]. We use vector packing to encode bids/votes into HTLPs and use custom zero-knowledge proofs (ZKPs) suited to the HTLP setting to enforce submission correctness. Besides simple protocols like first-past-the-post (FPTP) voting, we show how to realize more complicated auctions and elections, e.g., cumulative voting. We define a syntax and security properties for time-locked voting and auction protocols and prove the security of Cicada with respect to these definitions.

Implementation Cicada is the first protocol for efficient, trustless, non-interactive, and fair auction/voting protocols. Despite the previous existence of our building blocks (HTLPs, ZKPs, and vector packing), combining these primitives in a way that maintains practical efficiency is non-trivial and has been a stumbling block for deployment. Thus, we also view our implementation of Cicada and the analysis of optimal deployment choices as a central contribution of this work.

We provide open-source implementations tailored to the popular Ethereum Virtual Machine (EVM) with a word size of 256 bits. Our protocols are particularly well-suited to the EVM since, unlike prior works, Cicada is non-interactive and we do not need to keep bids, ballots, and proofs in persistent storage as they are not required for any subsequent round. Our most efficient realizations work in $\mathbb{Z}_N^*$ for $N \approx 2^{1024}$, groups which are not natively supported by the EVM. We therefore also implement several gas-efficient libraries to support modular arithmetic in such groups of unknown order which may be of independent interest. We demonstrate in Section 5.6 that our protocols can be run today directly on Ethereum and cost only several USD on popular layer-2 solutions.

[1] Adult North American cicadas emerge from the ground at predictable intervals of 13 or 17 years. Similarly, our ballots/bids remain hidden only for a fixed interval.

Non-goals As discussed, our protocols target fairness via tally-privacy. Thus we view everlasting privacy as a non-goal. Nonetheless, we observe that Cicada is not inherently at odds with it and outline how it could be added to the framework in Section 5.7.1. Relatedly, we consider anonymity an orthogonal problem. Where desired, users can add anonymity via privacy-enhancing overlays [PSS19, Eth19, Nef01].

The voting literature also often views coercion-resistance [JCJ05], i.e., an adversary's inability to coerce voters to cast specific ballots, as a crucial property. There are two main pathways to obtaining coercion-resistance: receipt-freeness [BT94] or allowing unlikable revotes [LQT20]. Similar to previous work [Adi08], we see receipt-freeness as a desideratum in high-stakes democratic elections but view it as a non-goal in our protocol design and target low coercion-risk settings. Still, we note that coercion-resistance can be added to Cicada via unlinkable revotes: if submitted ballots are stored in a zero-knowledge set [Eth19], any ballot in the set can be revoked by (unlinkably) revealing its nullifier and adding it to an on-chain nullifier accumulator [Pri].

5.3 Additional Preliminaries

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.

5.3.1 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 5.2. We will show how to realize them in the Cicada framework in Section 5.5.

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.

	Submission domain	Hamming wt	Norm
FPTP voting	$[0,1]^m$	≤ 1	≤ 1
Approval voting	$[0,1]^m$	$\leq m$	$\leq m$
Range voting	$[0,w]^m$	$\leq m$	$\leq wm$
Cumulative voting	$[0,w]^m$	$\leq m$	$\leq w$
Ranked-choice voting (Borda)	$\pi([0,m-1])$	$m-1$	$m(m-1)/2$
Quadratic voting	$[0,\sqrt{w}]^m$	$\leq m$	$\|\mathbf{b}\|_2^2 = \langle \mathbf{b}, \mathbf{b} \rangle = w$
Single-item sealed-bid auction	$[0,w]$	1	$\leq w$

Table 5.2: Requirements for the domain, Hamming weight, and norm of a vector $\mathbf{b}$ 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 that can be assigned to any candidate.

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 5.3.3).

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 5.6). 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.

5.3.2 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 26 (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:*

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

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 the puzzle obtained by homomorphically applying the circuit C to m puzzles 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 $f(Z_1, \ldots, Z_m)$.

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 [Pai99], 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 5 (Linear HTLP [MT19])**.**

- $\underline{\mathsf{HTLP.Setup}(1^\lambda, T) \xrightarrow{\$} \mathsf{pp}}$*: Sample a strong semiprime N and a generator $g \leftarrow\!\!\$\ \mathbb{Z}_N^*$, 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 $Z = (u, v) \leftarrow \mathsf{HTLP.Gen}(\mathsf{pp}, s)$,

$$\mathsf{HTLP.Open}(\mathsf{pp}, Z) = \frac{(v/(h^R)^N \mod N^2) - 1}{N} = \frac{((1+N)^s) - 1}{N} = s \quad (5.1)$$

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 $i \in \{1, 2\}$,[2]

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

which opens to $f(s_1, s_2)$ by Equation (5.1).

Construction 6 (Multiplicative HTLP [MT19])**.**

- $\underline{\mathsf{HTLP.Setup}(1^\lambda, T) \xrightarrow{\$} \mathsf{pp}}$*: Same as Construction 5.*

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

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

[2] 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).

- HTLP.Open(pp, $Z, r) \rightarrow s$: *Parse* $Z := (u, v)$ *and compute* $w := u^{2^T} \mod N = h^r$ *via repeated squaring. Output* $s := v/w$.

- HTLP.Eval(pp, $f, Z_1, Z_2) \rightarrow Z$: *To evaluate a multiplicative function* $f(x_1, x_2) = ax_1x_2$ *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 6 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 $f(x_1, x_2) = ax_1x_2$ and all $Z_i = (u_i, v_i) \in \mathsf{Im}(\mathsf{HTLP.Gen}(\mathsf{pp}, s_i; r_i))$ for $i \in \{1, 2\}$,

$$\begin{aligned}
\mathsf{HTLP.Eval}(\mathsf{pp}, f, Z_1, Z_2) &= (u_1 \cdot u_2 \mod N, a \cdot v_1 \cdot v_2 \mod N) \\
&= (g^{r_1} g^{r_2} \mod N, \quad h^{r_1} h^{r_2} \cdot as_1s_2 \mod N) \\
&= (g^{r_1+r_2} \mod N, \quad h^{r_1+r_2} \cdot as_1s_2 \mod N) \\
&= \mathsf{HTLP.Gen}(\mathsf{pp}, f(s_1, s_2); r_1 + r_2)
\end{aligned}$$

As an alternative, we introduce a novel linear HTLP based on the exponential ElGamal cryptosystem [CGS97] over a group of unknown order (Construction 7). This can be seen as a lifting of the multiplicative HTLP to put s in the exponent, with changes shown in blue. The construction requires a small solution space $\mathcal{X} \subset \mathbb{Z}_N$, i.e., $\mathcal{X} = \{s : s \in \mathbb{J}_N \wedge s \ll N\}$.

Construction 7 (Efficient linear HTLP.)**.**

HTLP.Setup$(1^\lambda, T)$ $\$\!\!\rightarrow$ p**.** *Output* $\mathsf{p} := (N, g, h, y)$*, where* $y \leftarrow\!\!\$\ \mathbb{Z}_N^*$ *and the remaining parameters are the same as in constructions 5 and 6.*

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

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

HTLP.Open(p, $Z, r) \rightarrow s$**.** *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*.*

HTLP.Eval(p, $f, Z_1, Z_2) \rightarrow Z$**.** *To evaluate a linear function* $f(x_1, x_2) = b + a_1x_1 + a_2x_2$ *homomorphically on puzzles* $Z_1 := (u_1, v_1)$ *and* $Z_2 := (u_2, v_2)$*, return*

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

This scheme is only practical for small $\mathcal{X}$ 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 5.6.1.

Non-malleability. Introducing a homomorphism raises the issue of puzzle malleability, i.e., "mauling" one puzzle (whose solution may be unknown) into a puzzle with a related solution. This could lead to issues when HTLPs are deployed in larger systems, prompting the definition of non-malleability for TLPs [FKPS21]. We instead define and enforce non-malleability at the system level (see Section 5.4).

5.3.3 Vector Packing

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. This motivates the following definition.

Definition 27 (Packing scheme)**.** *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:*

- $\underline{\mathsf{PSetup}(\ell, w) \rightarrow \mathsf{pp}}$*: Given a vector dimension* ℓ *and maximum entry* w*, output public parameters* pp*.*
- $\underline{\mathsf{Pack}(\mathsf{pp}, \mathbf{a}) \rightarrow s}$*: Encode* $\mathbf{a} \in (\mathbb{Z}^+)^\ell$ *as a positive integer* $s \in \mathbb{Z}^+$*.*
- $\underline{\mathsf{Unpack}(\mathsf{pp}, s) \rightarrow \mathbf{a}}$*: 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 8 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 8 (Packing from Positional Numeral System)**.**

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

We also introduce an alternative approach in Construction 9 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 9 (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 PNS, which is only additively homomorphic for SIMD (single instruction, multiple data), 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 $\mathbf{a}_\times = \prod_{i\in[n]} \mathbf{a}_i$. 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 hence be larger as well. Although the RNS has found application in error correction [TC14, KPT+22], 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 show in Section 5.5.2 that RNS is a natural fit for some voting schemes, e.g., quadratic voting, leading to more efficient proofs of ballot correctness.

5.4 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 FPTP voting, the tally function t is addition, and the finalization function f is $\arg\max$ over the final tally/bids.

Definition 28 (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* $f : \mathcal{X}' \to \mathcal{Y}$*.*

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

$\mathsf{Seal}(\mathsf{pp}, i, s) \overset{\$}{\to} (\mathcal{Z}_i, \pi_i)$**.** *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 aggregate* $\mathcal{Z}$*, a time-locked submission* $\mathcal{Z}_i$ *of user* i*, and a proof* π_i*, the aggregation algorithm checks the proof and potentially uses* $\mathcal{Z}_i$ *to update* $\mathcal{Z}$ *to* $\mathcal{Z}'$*.*

$\mathsf{Open}(\mathsf{pp}, \mathcal{Z}) \to (\mathbf{s}, \pi_{\mathsf{Open}})$. *Using T sequential steps, open the time-locked aggregate $\mathcal{Z}$ to $\mathbf{s}$ (in a correct execution, $\mathbf{s} = t(s_1, \ldots, s_n)$) and compute a proof π_{Open} to prove correct opening of $\mathcal{Z}$ to $\mathbf{s}$.*

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

Note that the $\mathsf{Setup}(\cdot)$ algorithm may use private randomness. In particular, our implementation of Cicada uses cryptographic groups (RSA and Paillier groups) that cannot be efficiently instantiated without a trusted setup (an untrusted setup requires gigantic moduli [San99]). This trust can be minimized by generating the group via a distributed trusted setup, e.g., [BF01, CHI+21, DM10]. Alternatively, Cicada can be instantiated in a fully trustless manner using HTLPs over class groups [TCLM21], which do not require a trusted setup; however, class group HTLPs are less efficient and verifying them on-chain is more costly (see Section 5.6.1).

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 29 (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 $s_1, \ldots, s_n \in \mathcal{X}$,*

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

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

Submission privacy. The protocol 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 30 (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*

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

Non-malleability. 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, an attacker can always win by homomorphically computing an HTLP containing the sum of all the other participants' bids plus a small value ε. Therefore, we also define *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 31 (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*

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

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.*

Notice that the check $\mathsf{Aggr}(\mathsf{p}, \mathcal{Z}, i, \mathcal{Z}_i, \pi_i) \neq \mathcal{Z}$ captures the requirement that the proof π_i output by $\mathcal{A}$ should verify, since otherwise Aggr does not update the tally HTLP(s) but outputs the same $\mathcal{Z}$ as in the input.

5.5 The Cicada Framework

We present Cicada, our framework for non-interactive private auctions/elections, in Figure 5.2. 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 5.3.2), a vector packing scheme (Section 5.3.3), and matching NIZKs (which we present in Section 5.5.2) to ensure correctness of submissions by proving both the well-formedness of the puzzle and the solution's membership in $\mathcal{X}$.

We envision three types of participants, as illustrated in Figure 5.1:

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). 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.

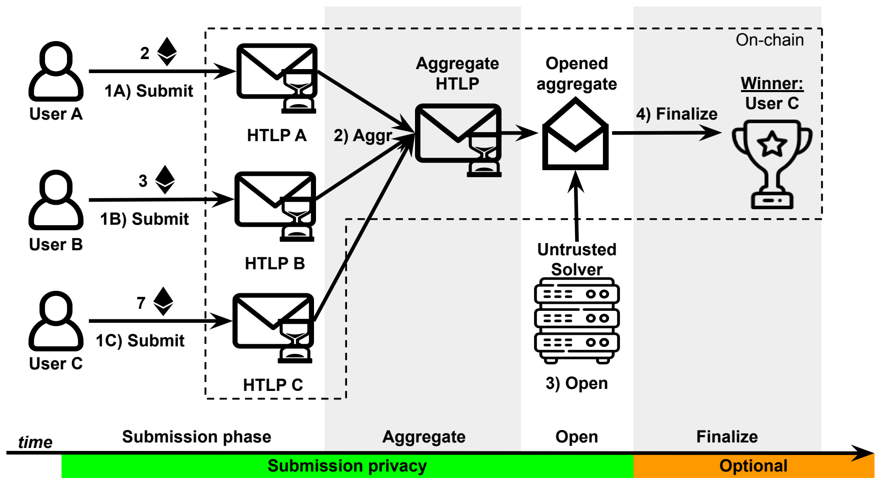

Figure 5.1: The system model of Cicada. *(1) Submission phase:* users generate their bids/ballots as HTLPs and post them to a public bulletin board, e.g., a blockchain. *(2) Aggregation:* an on-chain contract homomorphically combines submissions into an aggregate puzzle as they are submitted. *(3) Opening:* after all 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 5.7.1, we show how voters can pool their submissions to achieve everlasting ballot privacy.

Off-chain solver. We assume an untrusted *solver* who unlocks the final (set of) HTLP off-chain and submits the solution(s) to the coordinator with a proof of correct opening. In principle, this could be any party, but in practice will likely be one of the parties participating in (or administering) the vote/auction or a paid marketplace [Aba23, TGB$^+$21].

Cicada implements a time-locked auction/voting scheme (Definition 28) as follows: (0) System participants agree on a delay parameter T and packing parameter ℓ. The latter introduces a crucial design choice, defining a storage-computation trade-off we detail in Section 5.6. The coordinator (or a set of trusted parties, depending on the HTLP construction used) runs the Setup algorithm to output HTLP public parameters pp and initialize a (set of) aggregate HTLP(s) $\mathcal{Z}$ containing zeros. (1) A user i runs 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 $\mathcal{Z}_i$ is well-formed, i.e., its contents are in the domain $\mathcal{X}$ of the scoring function Σ. Users send their submissions and proofs to the on-chain coordinator. (2) Upon receiving a user submission $\mathcal{Z}_i, \pi_i$, the coordinator (Cicada smart contract) runs Aggr to verify π_i, and if it is valid,

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 5.5.2) and PoS be a proof of correct HTLP solution (see Section 5.5.2).

$\mathsf{Setup}(1^\lambda, T, \ell) \xrightarrow{\$} (\mathsf{pp}, \mathcal{Z})$. Set up the public parameters $\mathsf{pp}_{\mathsf{NIZK}} \xleftarrow{\$} \mathsf{NIZK.Setup}(1^\lambda)$, $\mathsf{pp}_{\mathsf{tlp}} \xleftarrow{\$} \mathsf{HTLP.Setup}(1^\lambda, T)$, and $\mathsf{pp}_{\mathsf{pack}} \leftarrow \mathsf{PSetup}(\ell, w)$. Let $\mathcal{Z} = \{Z_j\}_{j \in [m/\ell]}$ where $Z_j \xleftarrow{\$} \mathsf{HTLP.Gen}(0)$. 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 $\mathsf{NIZK.Verify}((i, \mathcal{Z}_i), \pi_i) = 1$, 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 $(\mathcal{S}, \pi_{\mathsf{open}})$.

$\mathsf{Finalize}(\mathsf{pp}, \mathcal{Z}, \mathcal{S}, \pi_{\mathsf{open}}) \to \{y, \bot\}$. If $\mathsf{PoS.Verify}(\mathcal{S}, \mathcal{Z}, 2^T, \pi_{\mathsf{open}}) \neq 1$, return $\bot$. 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 5.2: The Cicada framework for non-interactive private auctions and elections.

aggregates $\mathcal{Z}_i$ into the tally HTLPs $\mathcal{Z}$, resulting in updated tally $\mathcal{Z}'$. (3) 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) $\mathbf{s}$ of the tally HTLP(s) $\mathcal{Z}$ along with a proof of correct opening π_{open} (using a proof of solution PoS, see Section 5.5.2). The off-chain solver will send $\mathbf{s}, \pi_{\mathsf{open}}$ to the contract. (4) The on-chain contract runs $\mathsf{Finalize}$ to verify the correctness of $\mathbf{s}$ by checking π_{open}. If the check passes, it computes the auction/election winner(s) as $y = f(\mathbf{s})$.

Additive voting Cicada captures "additive" voting schemes, 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. This includes many popular voting schemes such as 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 entries, norm, etc.; see Table 5.2). Thus, Cicada can instantiate fair voting protocols for all these schemes.

Sealed-bid auctions The Cicada framework can also be used to implement a sealed-bid auction with a number of HTLPs which is independent of the number of participants n. Assuming bids are bounded by M, we use an HTLP with solution space $\mathcal{X}$ such that $|\mathcal{X}| > M^n$. Each user i submits $Z_i \leftarrow \mathsf{HTLP.Gen}(bid_i)$ and π_i, where π_i proves $0 \leq bid_i \leq M$. A packing of the bids is computed at aggregation time, with Aggr updating Z to $Z \boxplus (M^{i-1} \cdot Z_i)$. After the bidding phase, the final "tally" is opened to s^* and the bids are recovered as $\mathsf{Bids} := \{s^* \bmod M^{i-1}\}_{i \in [n]}$. Any payment and allocation function can now be computed over the bids; in the simplest case, the winner is $\arg\max_i(\mathsf{Bids})$ and their payment is $\max_i(\mathsf{Bids})$. Here, the full set of bids is revealed after the auction concludes since $\max_i$ is a nonlinear function and cannot be computed homomorphically using linear HTLPs.

Locking up collateral is necessary for every (private) auction scheme. We treat the problem of collateral lock-up as an important but orthogonal problem and refer to [TAF$^+$23] for an extensive discussion.

5.5.1 Security Proof of Cicada

Intuitively, submission privacy follows from the security of the HTLP (assuming the delay T is longer than the submission phase) and the zero-knowledge property of the NIZKs: 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.

Theorem 4. *Given a linear scoring function Σ, a secure NIZKPoK* NIZK *and proof of solution* PoS*, a secure* HTLP*, and a packing scheme* $(\mathsf{PSetup}, \mathsf{Pack}, \mathsf{Unpack})$*, the Cicada protocol Π_Σ (Figure 5.2) 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 29) 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 $\mathcal{A}$, cf. Definition 30. 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.

$\underline{\mathcal{H}_3}$: 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 (Definition 31). 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 $\mathsf{NIZK.Verify}((i, \mathcal{Z}_i), \pi) = 1$ 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 $\mathsf{NIZK.Verify}((i, Z_i), \pi_i) = 1$. 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. □

5.5.2 Ballot/Bid Correctness Proofs

For our NIZKs, 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 5), $\mathbb{G}_1 = \mathbb{J}_N$, $\mathbb{G}_2 = \mathbb{Z}^*_{N^2}$, $h = (g^{2^T})^N$, and $y = 1 + N$. For the exponential ElGamal HTLP (Construction 7), $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{Z}^*_N$, $h = g^{2^T}$, 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:

$$\mathcal{R}_{\mathsf{PoS}} = \{((y, u, v, w \in \mathbb{G}, s \in \mathbb{Z}); \perp) : w = u^{2^T} \wedge \ v = wy^s \in \mathbb{G}\} \tag{5.2}$$

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:

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

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.[3]

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 $s \mod p_j \equiv b_j \ \forall j \in [m]$, and that this encoding is fully SIMD homomorphic. It follows that $s^2 \mod p_j \equiv b_j^2$ for all $j \in [m]$.[4] With the RNS packing it therefore suffices to prove a square relationship *once* for the puzzles encoding s and s^2 (e.g., using $\mathsf{zk\text{-}PoKSqS}$) 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., $a_{ij} = s_i^2 \mod p_j$, with a Pedersen commitment, and use a variant proof of knowledge of exponent modulo p_j ($\mathsf{PoKEMon}$ [BBF19]) to show the commitments contain the appropriate values a_{ij}. Then, it can open the sum of the commitments. $\mathsf{PoKEMon}$ proofs are batchable, so the contract can verify them efficiently and check that the sum of the commitments opens to w.

5.5.3 Security Proofs of Sigma Protocols

Finally, we prove special-soundness and honest-verifier zero-knowledge (HVZK) of the sigma protocols in Section 5.5.2. Any such protocol can be made into a non-interactive zero-knowledge proof of knowledge (NIZKPoK) via the Fiat-Shamir transform [FS87].

[3] 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.

[4] 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 $s_j^2 \leq w < nw$.

Theorem 5 (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

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

Combining the two equations we get

$$\begin{aligned} h^{r\Delta e} y^{s\Delta e} &= h^{\Delta w} y^{\Delta x} \\ \iff \qquad v^{\Delta e} &= h^{\Delta w} y^{\Delta x} \end{aligned} \tag{5.3}$$

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 (5.3), 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

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

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 6 (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 5, 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. $\square$

Theorem 7 (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. $\square$

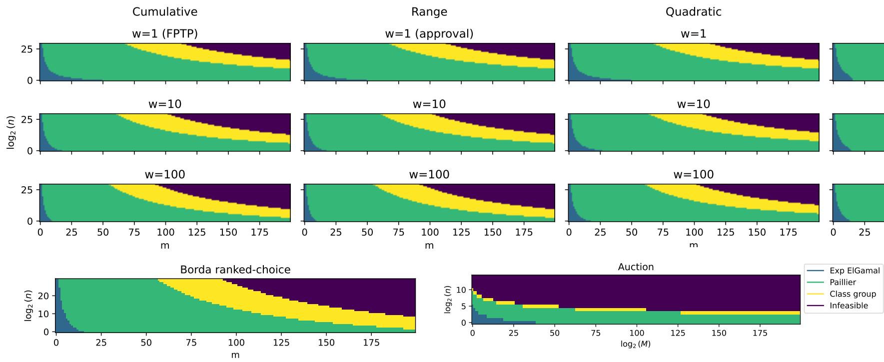

Figure 5.3: Most efficient HTLP construction for voting and auction using Cicada with maximal packing (using PNS except where indicated).

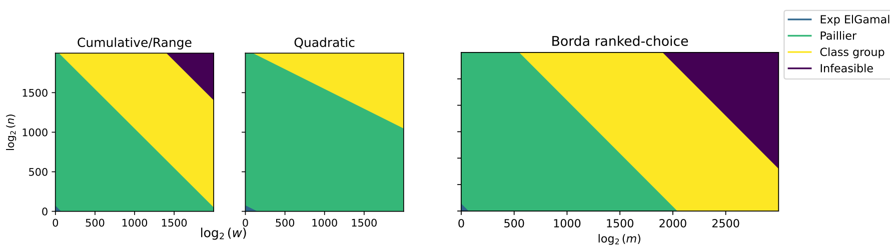

Figure 5.4: Most efficient HTLP construction for voting schemes using Cicada without packing. For the sealed-bid auction, the bid bit-length directly determines the HTLP construction to use (exponential ElGamal up to 80 bits, then Paillier up to 2048, and class group up to 3400).

5.6 Implementation

5.6.1 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 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 5.3 (with packing) and Figure 5.4 (without packing). We use the security parameter $\lambda = 80$ (see discussion in Section 5.6.2), 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 7) 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 5) 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.

5.6.2 Our Implementation

We implemented our transparent on-chain coordinator as an Ethereum smart contract in Solidity.[5] 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 Section 5.6.3) 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:

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

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.

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 Section 5.6.3, 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 Section 5.5.2.

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 $\mathsf{zk\text{-}PoKS}$) 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 ($\mathsf{zk\text{-}PoPS}$) via Legendre's three-square decomposition theorem [Gro05]. As a building block, we use a proof of square solution ($\mathsf{zk\text{-}PoKSqS}$) 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 Section 5.5 (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 $\mathsf{zk\text{-}PoKS}$ and two $\mathsf{zk\text{-}PoPS}$ 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.

	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 5.3: 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.

5.6.3 Deployment Costs

We report EVM gas costs of several instantiations of Cicada in Table 5.3.

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., $\ell = 1$). Submitting a binary vote ballot costs $418,358$ gas ($\approx$ 15.36 USD on Ethereum).[6] For cumulative voting, the submission cost scales linearly in m: with $m = 2$ candidates, submitting a ballot costs $3,391,514$ gas ($\approx$ 124.51 USD), and each additional candidate adds $\approx 1,699,847$ gas ($\approx$ 62.40 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 ($\approx$ 131.65 USD). 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 $(g^r)^{2^T}$. With exponential ElGamal, solving the puzzle also requires the off-chain solver to brute-force a discrete logarithm. The correctness of this computation is proven to the contract with Wesolowski's PoE [Wes19] (recalled in Section 5.5.2). The Finalize cost comes from verifying the PoE(s) on-chain, which burns $101,066$ gas ($\approx$ 3.71 USD) per proof. Without packing, the untrusted solver must provide a Wesolowski proof per tally HTLP,

[6] 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.

so the Finalize gas cost is linear in the number of tally HTLPs. A portion of the Wesolowki verification cost comes from checking that the challenge is a prime number. Our implementation uses a Baillie-PSW [PSW80] primality test, which costs $44,972$ gas (≈ 1.65 USD).

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 5.4: EVM gas costs of verification for the proof systems described in Section 5.5.2.

Verification costs Our Solidity implementation includes the sigma protocols described in Section 5.5.2. We report their verification costs in Table 5.4. With Groth's trick [Gro05] in the proof of positivity (zk-PoPS), we must decompose the integer solution into 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).

	Binary vote	**Sealed-bid auction** ($\ell = b$)
	$m = 1$	$b =$ any
Aggr (ETH L1)	15.36	112.16
Aggr (Arbitrum Nova)	0.35	2.55
Aggr (Optimism)	0.009 18	0.067 01
Aggr (zkSync Era)	0.063	0.459
Finalize (ETH L1)	4.25	5.42
Finalize (Arbitrum Nova)	0.10	0.12
Finalize (Optimism)	0.002 54	0.003 24
Finalize (zkSync Era)	0.017	0.022

Table 5.5: USD costs of Cicada binary vote and fully packed ($\ell = b$) sealed-bid auction on Ethereum L1, Arbitrum Nova, Optimism, and zkSync Era as of July 30, 2024 at 7:30 PM UTC.

USD cost estimates on L1 and L2 In the short-term, deploying on Layer 2 (L2) already brings the costs down by 1–2 orders of magnitude. In Tables 5.5 and 5.6, we provide a rough estimate of the concrete cost in USD as of July

	Cumulative vote ($\ell = 1$)				
m	2	3	4	5	6
Aggr (ETH L1)	124.51	186.55	248.95	311.67	374.75
Aggr (Arbitrum Nova)	2.83	4.24	5.66	7.08	8.52
Aggr (Optimism)	0.074 39	0.111 45	0.148 74	0.186 21	0.223 90
Aggr (zkSync Era)	0.509	0.763	1.018	1.275	1.533
Finalize (ETH L1)	9.89	14.60	19.16	23.67	28.30
Finalize (Arbitrum Nova)	0.22	0.33	0.44	0.54	0.64
Finalize (Optimism)	0.005 91	0.008 72	0.011 45	0.014 14	0.016 91
Finalize (zkSync Era)	0.040	0.060	0.078	0.097	0.116
	Sealed-bid auction ($\ell = 1$)				
b	8	10	12	14	16
Aggr (ETH L1)	131.65	164.76	198.02	231.43	265.01
Aggr (Arbitrum Nova)	2.99	3.74	4.50	5.26	6.02
Aggr (Optimism)	0.078 65	0.098 44	0.118 31	0.138 27	0.158 33
Aggr (zkSync Era)	0.539	0.674	0.810	0.947	1.084
Finalize (ETH L1)	36.90	46.00	54.98	64.23	73.54
Finalize (Arbitrum Nova)	0.84	1.05	1.25	1.46	1.67
Finalize (Optimism)	0.022 05	0.027 48	0.032 85	0.038 37	0.043 94
Finalize (zkSync Era)	0.151	0.188	0.225	0.263	0.301

Table 5.6: USD costs of unpacked ($\ell = 1$) Cicada cumulative voting and sealed-bid auctions for various settings of m, b on Ethereum L1, Arbitrum Nova, Optimism, and zkSync Era as of July 30, 2024 at 7:30 PM UTC.

30, 2024 of deploying Cicada on Ethereum and several Layer-2 (L2) networks. These numbers are conversions of the gas costs in Table 5.3 and should not be viewed as precise costs predictions, but rather as evidence of Cicada's concrete efficiency and feasibility. Precisely benchmarking costs in terms of USD is difficult due to the volatility of both the ETH/USD price and of transaction/priority fees on Ethereum and its L2s. At the time of conversion (July 30, 2024 at 7:30 PM UTC), the Ethereum price was 0.00000333736 USD per gwei and a medium priority fee on Ethereum L1 was 11 gwei. We also consider three popular Ethereum rollups: Arbitrum Nova, Optimism, and zkSync Era. At the time of our measurement, transaction fees for Arbitrum Nova, Optimism, and zkSync Era were 0.25, 0.006572 and 0.045 gwei, respectively.

While possible, our Cicada deployments are overall prohibitively expensive on Ethereum L1. However, the costs are quite reasonable on its L2s: participating in a Cicada sealed-bid auction, cumulative vote, or binary vote would cost less than 1 USD on these popular Ethereum L2s, which we deem highly practical. 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.

5.7 Extensions

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

5.7.1 Everlasting Ballot Privacy for HTLP-based Protocols

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 5.2, 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 at the cost of additional *off*-chain interaction.

Observe that all known *efficient HTLP constructions* are of the form $(u, v) = (g^r, h'^r X)$,[7] where the solution is encoded in X and recovering it requires 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 aggre-

[7] In the exponential ElGamal case, $h' = h$, while in the Paillier construction, $h' = h^N$ (see Section 5.3.2). 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 a $k-1$ 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 5.5: The on- and off-chain ballot batching protocols that $k < n$ parties can use to achieve everlasting ballot privacy.

gate 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 5.5 for details).

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.

5.7.2 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 5.6). 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:

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 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 $Z_{\mathsf{sk}} \xleftarrow{\$} \mathsf{TLP}.\mathsf{Gen}(\mathsf{pp}_{\mathsf{tlp}}, \mathsf{sk})$ and return $\mathsf{pp} := (\mathsf{pp}_{\mathsf{tlp}}, \mathsf{pk})$ and $\mathcal{Z} := (Z_{\mathsf{sk}}, \varnothing)$.

$\mathsf{Seal}(\mathsf{pp}, i, s) \xrightarrow{\$} (\mathsf{ct}_i, \pi_i)$**.** 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.

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

$\mathsf{Open}(\mathsf{pp}, \mathcal{Z}) \to (\mathsf{sk}, \pi_{\mathsf{open}})$**.** Parse $\mathcal{Z} := (Z_{\mathsf{sk}}, \mathcal{C})$. Compute and output $\mathsf{sk} \leftarrow \mathsf{HTLP}.\mathsf{Solve}(\mathsf{pp}_{\mathsf{tlp}}, Z_{\mathsf{sk}})$ and $\pi_{\mathsf{open}} \leftarrow \mathsf{PoS}.\mathsf{Prove}(\mathsf{sk}, Z_{\mathsf{sk}}, 2^T)$.

$\mathsf{Finalize}(\mathsf{pp}, Z_{\mathsf{sk}}, \mathsf{sk}, \pi_{\mathsf{open}}) \to \{y, \bot\}$**.** If $\mathsf{PoS}.\mathsf{Verify}(\mathsf{sk}, Z_{\mathsf{sk}}, 2^T, \pi_{\mathsf{open}}) \neq 1$, return $\bot$. Otherwise, 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 5.6: The "solve one, get many for free" paradigm (CJSS) [CJSS21].

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.

Chapter 6

High-Value Secret-Key Backups[§]

Contents

Reliable storage of cryptographic secret keys is challenging. This situation is particularly exacerbated in the cryptocurrency ecosystem, where anyone who controls the signing key for an account can typically take arbitrary action on the user's behalf. For example, an attacker with access to a user's key could transfer large sums of money out of a user's account with no recourse. Numerous

[§] *Portions of this section have been adapted from [GGJ$^+$24].*

cryptocurrency wallets have been developed to allow users to safeguard their keys while preserving their ease of use; we next describe some popular types.

Threshold wallets Cryptocurrency wallets need to overcome significant technical hurdles in their quest to increase reliability while preserving the security of the keys. For instance, attempts at increasing reliability using replication worsen security. This is because any replication makes it easier for the attacker to potentially access one of the replicated copies of the key. Threshold secret-sharing systems [Sha79] offer an elegant middle ground: user keys are secret-shared among n parties which we refer to as *custodians*, and access to t of them is required to access the key. Thus, user keys can be recovered even if $n - t$ of the custodians become non-responsive. At the same time, an attacker needs to to access t key shares to recover the key.

Custodian threshold wallets offer robust security and reliability guarantees. However, these custodians need to remain online to provide users with easy access to their keys. For example, at least t out of n custodians need to remain online at all times in case a user decides to perform a transaction involving a secret key that the custodians hold on the user's behalf. Furthermore, this requirement that the custodians are always online poses additional risks. In particular, a software vulnerability in the custodian servers could jeopardize the secrecy of the held secret shares, and such a vulnerability could be easy to exploit since these servers are necessarily online and listening for requests.

Cold wallets Cold (i.e., offline) wallets avoid the aforementioned limitation of the custodian hot (i.e., always online) wallets because they do not always stay online. However, this affects their usability. A typical compromise 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 keypair. 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].

Backing up high-value keys While hot and cold wallets provide reasonable security and usability tradeoffs, all of the systems built upon them today seem inadequate for backing up rarely-used high-value secret keys. In particular, users or institutions may want to securely and reliably store high-value asset keys they don't frequently need access to. Furthermore, users may want to back up keys from other systems for recovery in case of catastrophic losses in the parent system. This is an important use case that arises in several situations. In particular:

1. Consider a cryptocurrency exchange which is frequently used by individuals to store cryptocurrency long-term, in a similar way to a bank [Arm16,

Coi, Kra]. The net inflows and outflows of cryptocurrency from the exchange might be roughly equal over short- and mid-term periods, meaning that the exchange rarely needs to access the bulk of its cryptocurrency deposits. However, because the value of these deposits could be extremely large, the key (or keys) safeguarding these rarely-accessed deposits need to be securely backed up in a way that reflects their value.

2. An individual with a high cryptocurrency net worth would be in a very similar situation: they might secure the bulk of their cryptocurrency with a key that is very rarely used and perhaps send enough funds once a year to a "hot" wallet or similar system for their yearly expenses.

3. Finally, interestingly, this issue also arises in the threshold wallet setting. Many systems may remove threshold signing parties when they fail to meet some predefined criteria, e.g., exceed maximal response latency or produce invalid signature shares. However, to maintain high security, threshold wallets should maintain a minimum number of nodes. If the signing set becomes too small, parties can no longer be removed, but may still exhibit behavior that would warrant removal under normal circumstances. In such cases, a backup recovery process is needed to regenerate the signing key shares.

   For example, Lit Protocol, a cryptographic key management provider offering a threshold signing network, specifies a backup recovery mechanism to restore the system to an operational state in such cases [Lit24]. However, the backup system currently functions only as a snapshot and is not ideal as a backup system. In particular, the current backup system does not support refreshes of the key shares, reducing the security of the overall system, and has no way of checking backup integrity.

6.1 Novel Design Requirements

In our envisioned high-security backup system, cryptographic secret keys are used infrequently. This contrasts with the goals of easy accessibility in traditional wallet systems. Thus, our new setting necessitates a hardened system along with a novel set of design requirements aimed at defending against well-resourced nation-state adversaries. In particular:

1. **Hardened Security and Strong Recovery Properties.** Similar to threshold wallets, we want backed-up keys to be secret-shared and have the shares spread over a highly distributed network. First, we want geographic distribution to avoid key-share losses from natural disasters. Second, we need key shares placed in disparate jurisdictions to safeguard against government actors. Finally, we need key shares placed with machines deployed from different hardware manufacturers using varying operating systems. Given these more stringent requirements, storing key shares across more

custodians than is typical (e.g., 67-out-of-100) is warranted. This allows for hardened security and reliability in case recovery is needed.

2. **Security of Each Secret Share is Backed by a Cold Wallet Portion.** Online machines get hacked rather frequently. Furthermore, in the case of nation-state adversaries, the exploited vulnerability can be quite sophisticated and could remain undetected for years. Thus, we require that each custodian holds each user-key's secret share jointly in a hot (i.e., online) and a cold (i.e., offline) portion of the wallet. The cold wallet should always stay offline, potentially on an air-gapped computer in a secure location. Of course, the cold wallet will need to be accessed if recovery is initiated, but this is the only time it should be used. Thus, recovery will need access to t cold wallet portions, one each for each utilized secret share. Consequently, the recovery process will be slow. However, for us this is not a deal-breaker since in our system, recovery requests are infrequent. In fact, this additional lockup period provides time to perform due diligence in checking the veracity of the recovery request.

   Nonetheless, we insist that the backup process be fast and avoid the need for access to the cold portion of the wallet. Implicitly, this requires that no key-share-specific information is stored on the cold portion of the wallet, and its memory requirements are minimal — in particular, independent of the number of users whose keys it stores.

3. **Post-Compromise Security.** Given that online machines get hacked frequently, it is possible that over time, an attacker may gather cryptographic secrets held by several custodians. Thus, we require that the system supports hot wallet portions regularly updating their key material in coordination with the other hot wallet custodians such that infected machines can recover from such leakage.

4. **Continual Assurance of a Key's Safekeeping.** We also need a mechanism to continually assure users that their keys are safely stored. This feature is critical because our system is not designed to allow users to easily make transactions, which typically also serve as a way for users to check the safe storage of their keys. Absent such continual assurances, users could be duped into a false sense of security that their keys have been safely backed up. Thus, we require security against (potentially) malicious custodians who actively delete user keys while still attempting to falsely convince users about the safekeeping of their keys.

5. **Hiding System Users.** Finally, given a user transaction on chain, it should not be possible to determine whether it was created using our system. This is essential, since such information could help attackers identify and target users with high-value keys.

Lastly, we remark that any secret key storage solution is only as good as the strength of the keys it stores. Thus, we insist that the keys stored in our system are themselves generated via a distributed key generation (DKG) protocol, and the key shares are delivered to the custodians directly.

Going forward, we will refer to a backup system meeting the above design requirements as an *auditable hot-cold threshold backup*.

6.1.1 Our Contribution

Building on the design requirements stated above, we introduce a new wallet system called Throback ("THReshold Online/offline BACKups"). Throback is designed to serve as a high-security backup system for high-value cryptographic keys, simultaneously meeting all the outlined design requirements while offering high efficiency. Below, we describe in order how our construction addresses each of the design requirements.

1. **Non-interactive Protocol.** Throback requires almost no interaction between parties both for recovery and to verify the correct storage of key material. This allows it to efficiently scale to large, highly distributed networks of custodians. Furthermore, cold parties in Throback can even *generate* their key material independently, without interacting with the hot parties or with each other. This reduces the cold parties' attack surface even more.

2. **Hot-Cold Model.** Our construction distributes secret shares among *pairs* of hot and cold parties. In this way, an attacker must corrupt *both* the cold and hot components of a custodian 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: 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, and is therefore much more difficult to attack since the cold components are almost always offline.

   The cold parties in our construction only need to store a constant number of elements, which importantly is independent of the number of custodied secrets. This is particularly desirable since cold parties are normally resource-constrained devices such as hardware wallets.

3. **Proactive Refresh.** Our protocol allows periodic updates of the hot key shares. These share refreshes force an attacker to compromise at least t hot-cold pairs within a single *epoch* to exfiltrate the key: otherwise, any key material obtained is made obsolete by the refresh operation.

4. **Proofs of Remembrance.** We show how the hot and cold parties in our protocol can produce "proofs of remembrance", i.e., zero-knowledge proofs that they still possess their key material. Our proof systems must guarantee that the underlying secret key is still available and unchanged

while also accounting for changing values of the *individual* shares due to proactive refreshes. We show how to meet both needs simultaneously, allowing a client to intermittently and independently audit its (hot or cold) custodians. This allows the client to ensure that its key material has not been overwritten or forgotten, and its funds are still accessible.

5. **Threshold BLS Construction.** Throback produces standard BLS signatures so that it is not possible to identify users of the system (i.e., holders of high-value cryptographic keys). As we argued above, this is crucial for a system designed to protect high-value secrets.

We implemented Throback 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[1] and is Apache 2.0-licensed.

Open problems BLS signatures have the advantage of being simple to understand and deploy. This has led to their widespread use in production systems, including 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 can specify alternative signature schemes to verify their transactions, further easing the adoption of BLS-based wallets. However, 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 also popular threshold signature schemes in the literature. We leave the exciting problem of extending our results to these signature schemes open.

6.2 Technical Overview

Given a t-out-of-n sharing $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ of the secret key of a high-value signing keypair $(\mathsf{vk}, \mathsf{sk})$, we wish to store the shares among a set of hot-cold custodian pairs according to the requirements outlined in Section 6.1. A natural starting point for threshold signatures with the desired hot-cold access structure is to give each hot-cold pair a 2-out-of-2 (additive) sharing of one of the threshold shares. Even with this simple construction, our system model adds a hurdle to signing because cold parties can communicate only via their respective hot parties. This means that some care has to be taken in the signing procedure to ensure that *both* parties must participate for every signature. In particular, any communication from the cold party to the hot should be "bound" to the message m being signed (and some nonce). Otherwise, the hot storage could replay the communication from the cold party and produce a signature on some

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

other message $m' \neq m$ without the cold storage's cooperation, violating our threat model.

For BLS (threshold) signatures, this is not a problem, since signing is already non-interactive and the scheme's homomorphism means we can extend this to the hot-cold case as well. As described above, assume that the cold and hot parties hold shares $\mathsf{sk}_i^{\mathsf{cold}} := r$ and $\mathsf{sk}_i^{\mathsf{hot}} := \mathsf{sk}_i + r$, respectively, of the threshold signing key share. Then the cold party can send a partial signature $\sigma_i^{\mathsf{cold}} := H(m)^{\mathsf{sk}_i^{\mathsf{cold}}}$ to the hot party which is "bound" to the message m, and the hot party computes its own partial signature $\sigma_i^{\mathsf{hot}} := H(m)^{\mathsf{sk}_i^{\mathsf{hot}}}$. The threshold BLS signature σ_i can be computed by the hot party 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 party a t-out-of-n sharing of zero, which the latter adds to its current share.

Unfortunately, there are two main problems with this construction. First, it is unclear how to securely generate each hot-cold pair's additive sharing without involving a trusted third party or a costly distributed protocol. Assuming the key shares were previously generated via a DKG, the most straightforward option is for a trusted dealer to sample $\mathsf{sk}_i^{\mathsf{cold}}$ on behalf of each cold party and send it to them over a secure channel[2] (via the corresponding P_i^{hot}). However, to compute the hot shares $\mathsf{sk}_i^{\mathsf{hot}}$, this dealer must learn the current (potentially refreshed) values of sk_i, undermining their security. Alternatively, holders of the shares sk_i could engage in a computationally intensive and highly interactive protocol to generate the backup shares. A third option is to generate $\mathsf{sk}_i^{\mathsf{hot}}, \mathsf{sk}_i^{\mathsf{cold}}$ as part of the initial DKG, but this would preclude *ad hoc* backups of existing key shares. It would also require all custodians for the backup to be known from the start, ruling out dynamic joins. None of these approaches are desirable: ideally, generating a backup should require little or no interaction between custodians and avoid undermining the security of the high-value secret to be backed up.

The second problem with our strawman construction is enabling independent proofs of remembrance for hot and cold parties. In a simple t-out-of-n threshold signature, the secret key shares sk_i are Shamir shares [Sha79] of the secret key sk. Proving remembrance of these shares is relatively simple because they lie on a degree-$(t-1)$ polynomial: a natural approach for proving knowledge and well-formedness is for the dealer to publish a KZG commitment [KZG10] to the polynomial and distribute opening proofs for each value sk_i. To truly ensure remembrance of sk_i at any point in time, a party is required to provide a non-replayable proof; simply forwarding the proof given to it by the dealer should not suffice, since the party may have deleted the key share in the meantime. In fact, a party cannot respond directly with the KZG opening proof regardless, since the opening value—in this case sk_i—is required to verify the proof. To solve this issue, previous work [ZBK+22] showed how to provide blinded KZG evaluation proofs, also ensuring *knowledge* of the evaluation in the process.

[2] We assume cold parties register a public key with the dealer or some third-party service.

In our strawman construction, however, hot parties have shares $\mathsf{sk}_i^{\mathsf{hot}} := \mathsf{sk}_i + r_i$, which are no longer guaranteed to lie on a degree-$(t-1)$ polynomial. (The cold proofs are not a problem, since one can use a simple proof-of-knowledge of discrete logarithm to prove remembrance of a cold share r_i.) Forcing the $\mathsf{sk}_i^{\mathsf{hot}}$ to lie on a degree-$(t-1)$ polynomial by computing the cold values r_i as t-of-n shares is not an option, as it would require coordination among the normally offline cold parties and run into some of the same problems as hot-cold share generation above. This leaves hot parties with the task of proving knowledge of a value $\mathsf{sk}_i + r_i$ without knowing sk_i or r_i. Even given a KZG commitment and opening proof for sk_i along with the value g^{r_i}, it is unclear how a hot party could produce a proof of knowledge of the sum $\mathsf{sk}_i + r$ without relying again on a trusted setup with a static set of participants.

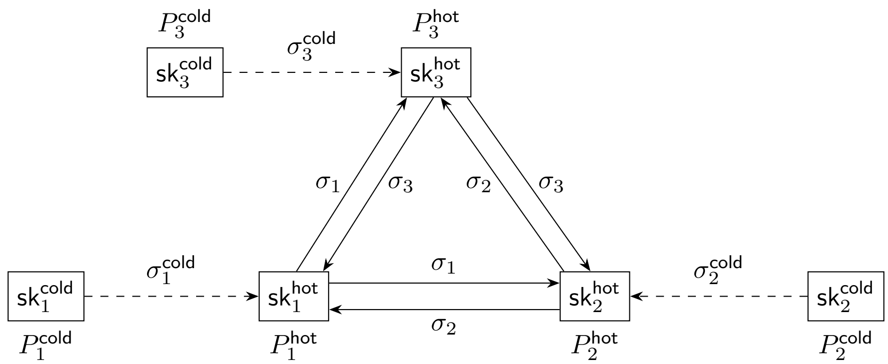

Figure 6.1: A auditable hot-cold threshold backup with $n = 3$. Given a message m, each P_i^{cold} uses its cold share $\mathsf{sk}_i^{\mathsf{cold}}$ to compute a cold partial signature σ_i^{cold} on m and sends it to its hot storage. P_i^{hot} uses m and its hot share $\mathsf{sk}_i^{\mathsf{hot}}$ to compute σ_i^{hot}, which it combines with σ_i^{cold} to get σ_i. The hot parties broadcast their partial signatures to each other to reconstruct the full signature σ on m. The dashed lines between each cold and hot storage represent the authenticated channel between them which is only active at signing time. The hot parties are always online and connected, represented by the solid lines.

Our approach It turns out that there is a way to realize this additive sharing without running into the issues above. Below, we show how to do this in a way that simultaneously enables both non-interactive generation of the backup shares and individual proofs of remembrance.

At a high level, each cold party will sample an encryption keypair $(\mathsf{ek}_i, \mathsf{dk}_i)$. The hot key shares will be encryptions of the secret key shares sk_i under a malleable encryption scheme. Malleability is needed so the underlying share can be refreshed even in encrypted form. In particular, we use an additive variant of ElGamal encryption [ElG84] where the message space is a field. This is necessary since our plaintext will be a field element (namely sk_i) and will be refreshed via addition to the ciphertext.

To make an additive variant of ElGamal over a field $\mathbb{F}$, consider the straightforward modification of ElGamal to use an additive instead of multiplicative mask by letting $\mathsf{Enc}(\mathsf{ek}, m) := (g^\rho, m + \mathsf{ek}^\rho)$. However, we said the plaintext m will be a field element, and ek is a group element. To enable addition, we will make use of an injective function $\mathcal{H} : \mathbb{G} \to \mathbb{F}$ and redefine $\mathsf{Enc}(\mathsf{ek}, m) := (g^\rho, m + \mathcal{H}(\mathsf{ek}^\rho))$. Encryption is still (additively) malleable, so we can add a shift $s \in \mathbb{F}$ to the message by adding it to the ciphertext, i.e., compute $\mathsf{Enc}(\mathsf{ek}, m + s)$ as $\mathsf{Enc}(\mathsf{ek}, m) + s$. In the context of the encrypted key share, this means the share sk_i being held in encrypted form can still be refreshed via addition: given a zero share z_i, refresh by computing $\mathsf{Enc}(\mathsf{ek}_i, \mathsf{sk}_i) + z_i = \mathsf{Enc}(\mathsf{ek}_i, \mathsf{sk}_i + z_i)$.

Returning to hot-cold shares as additive sharings of sk_i, let the hot party's share be the second element of a ciphertext encrypting sk_i under the cold party's key ek_i; then the cold party's share is the additive mask $\mathcal{H}(\mathsf{ek}_i^\rho)$. The pair's secret sk_i is recovered as $\mathsf{sk}_i^{\mathsf{hot}} - \mathsf{sk}_i^{\mathsf{cold}} = (m + \mathcal{H}(\mathsf{ek}_i^\rho)) - \mathcal{H}(\mathsf{ek}_i^\rho) = \mathsf{sk}_i$. Unfortunately, this construction still requires the cold party's storage to scale linearly with the number of secrets stored by the hot-cold pair. As stated in Section 6.1, we would like to make the cold storage independent of the number of user secrets.

We make one final change to the encryption scheme to accomplish this. Instead of sampling fresh randomness ρ for each ciphertext, ρ will be based on the secret sk being stored (the high-value signing key being backed up). In particular, we set the cold party's share (and the ciphertext mask) to $\mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}})$. Since the input is no longer uniformly random, the function $\mathcal{H}$ must be designed so its output is nevertheless close to uniformly random on high-entropy inputs (we give an efficient construction using the Leftover Hash Lemma in Section 6.4.1).[3] Furthermore, we instantiate the cold encryption keypairs in the same group and using the same generator as the signing keypair, so that $\mathsf{ek}_i^{\mathsf{sk}} = (g^{\mathsf{dk}_i})^{\mathsf{sk}} = \mathsf{vk}^{\mathsf{dk}_i}$. Now each cold party can store only a single element, namely dk_i, regardless of the number of clients. When it receives a signing request, it re-computes its share of sk_i on-the-fly as $\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i}) = \mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}})$.

A partial BLS signature σ_i can be computed by decrypting the hot key share "in the exponent": given m and vk, each hot party computes $\sigma_i^{\mathsf{hot}} := H(m)^{\mathsf{sk}_i^{\mathsf{hot}}}$ and the corresponding cold party computes $\sigma_i^{\mathsf{cold}} := H(m)^{\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})}$. The pair's partial signature σ_i is computed as $\sigma_i^{\mathsf{hot}} / \sigma_i^{\mathsf{cold}} = H(m)^{\mathsf{sk}_i}$, which is equal to a normal partial BLS signature under the ith signing key share.

Solving Problem 1: Generating hot-cold shares Now that hot shares are public-key encryptions of the secret shares, they can be computed without interacting with the cold parties. Given shares $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ of the high-value key, we can compute each hot share $\mathsf{sk}_i^{\mathsf{hot}} := \mathsf{sk}_i + \mathcal{H}(\mathsf{ek}_i^{\mathsf{sk}})$ by using the corresponding cold party's published encryption key.

[3] This actually requires us to expand the encryption key to 2 elements, which we ignore in the presentation here for simplicity. Please see the body of the paper for full details.

Solving Problem 2: Proofs of remembrance The independence between hot and cold parties also enables individual proofs of remembrance. Cold parties can use a standard proof of knowledge of discrete logarithm (for dk_i corresponding to ek_i), which suffices to prove the party can recover the cold share $\mathcal{H}(\mathsf{vk}^{\mathsf{dk}_i})$ for any verification key vk. As for hot parties, their shares still do not lie on a degree-$(t-1)$ polynomial, but they do lie on a degree-$(n-1)$ polynomial which the parties can interpolate at backup time. We therefore adapt the KZG-based proof idea to this degree-$(n-1)$ polynomial, with hot parties providing blinded evaluation proofs as their proofs of remembrance. The homomorphism of KZG commitments allows proactive refreshes, and we use a folklore approach to ensure the refreshes are of degree $(t-1) < (n-1)$.

6.2.1 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 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 (or signature shares).

6.3 Additional Preliminaries

A note on our ideal functionalities We model security in the UC framework (Section 2.8). As in some previous work [LN18, Kat24], our UC functionalities in this chapter contain some cryptographic operations specific to our construction. While this formulation is less general, it renders the analysis more straightforward and suffices for our purposes.

6.3.1 Leftover Hash Lemma

We use the presentation of the leftover hash lemma (LHL) [IZ89] from [AMPR19].[4] 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:

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

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

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

Lemma 7. (Leftover Hash Lemma.) *Let $(\mathcal{X}, \oplus)$ be a finite group, and let $\mathcal{S}_{\mathbf{x}}$ and $\mathcal{U}_{\mathcal{X}}$ be two distributions over $\mathcal{X}$ as defined above. For any (large enough) positive integer n, it holds that*

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

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

6.3.2 Model

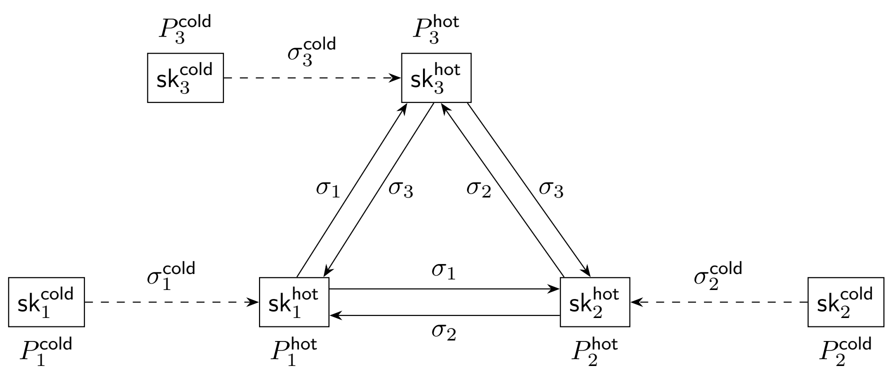

Figure 6.2: A auditable hot-cold threshold backup with $n = 3$. Given a message m, each P_i^{cold} uses its cold share $\mathsf{sk}_i^{\mathsf{cold}}$ to compute a cold partial signature σ_i^{cold} on m and sends it to its hot storage. P_i^{hot} uses m and its hot share $\mathsf{sk}_i^{\mathsf{hot}}$ to compute σ_i^{hot}, which it combines with σ_i^{cold} to get σ_i. The hot parties broadcast their partial signatures to each other to reconstruct the full signature σ on m. The dashed lines between each cold and hot storage represent the authenticated channel between them which is only active at signing time. The hot parties are always online and connected, represented by the solid lines.

We will use the superscripts hot and cold, respectively, to denote the hot and cold components of some value, and a subscript i to mean that 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.

Let $I_1, \ldots, I_n$ be parties (each representing a custodian institution) who will store shares of some user's signing key sk. Each institution I_i controls two parties: a hot wallet (P_i^{hot}) and a cold wallet (P_i^{cold}). Thus we represent an institution by the tuple $I_i = (P_i^{\mathsf{hot}}, P_i^{\mathsf{cold}})$. As in the standard threshold wallet setting, the hot parties are connected to each other via authenticated (but not private) channels. We also assume the parties can send broadcast messages (e.g. by posting to a blockchain). In contrast, each cold party is connected by an authenticated channel only to its corresponding hot party. This channel is only active during the signing phase, further reducing the cold party's attack surface. (For example, the cold party could be a read-only USB device which is plugged into a PC (the hot party) only briefly to produce a signature.) An illustration of this model is given in Figure 6.2 for $n = 3$.

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

6.4 Hot and Cold Key Shares

The core idea of our construction is for each hot party to store an encryption of the signing key share sk_i given to the custodian, with the corresponding decryption key kept in cold storage. That is, each custodian will generate an encryption keypair $(\mathsf{ek}_i, \mathsf{dk}_i)$ 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 BLS signature σ_i of m under sk_i.

Our construction uses a modified ElGamal [ElG84] encryption scheme to encrypt key shares. The homomorphic properties of the scheme will allow decryption of the key share "in the exponent" to obtain a message-specific 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 and $g \in \mathbb{G}$ is a generator. In our case, however, we will be encrypting secret key shares, which are field elements (specifically, elements of $\mathbb{Z}_p$).

Adapting ElGamal to plaintexts $m \in \mathbb{Z}_p$, we will compute a ciphertext as $(g^r, \mathcal{H}(\mathsf{ek}^r) + m) \in \mathbb{G} \times \mathbb{Z}_p$, where $\mathcal{H} : \mathbb{G} \to \mathbb{Z}_p$. Now m is masked with the uniformly random output of $\mathcal{H}$ and can be unblinded using dk by recomputing $\mathcal{H}(\mathsf{ek}^r) = \mathcal{H}((g^r)^{\mathsf{dk}})$. Like the original ElGamal encryption, this construction is malleable (in this case, additively rather than multiplicatively), which allows "shifting" of the message m in a ciphertext by an additive factor (made explicit via the Shift algorithm). We will use this property to enable proactive refresh of the hot (encrypted) key shares.

Construction 10 (additive ElGamal)**.** *Let $\mathbb{G}$ be a DLog-hard group of order p with generator g and $\mathcal{H} : \mathbb{G} \to \mathbb{Z}_p$.*

- $\underline{(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KGen}(1^\lambda)}$: *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* $ct' := (ct_0, ct_1 + \delta)$.

6.4.1 Additive Secret Sharing from Additive ElGamal

The Enc algorithm in Construction 10 takes an explicit randomness input r. In the context of our wallet construction, instead of sampling fresh randomness $r \in \mathbb{Z}_p$ for each hot party's ciphertext (hot key share), we will use a value based on the user secret being stored. In particular, when storing a signing keypair $(\mathsf{vk} := g^{\mathsf{sk}}, \mathsf{sk})$, we will use sk as the encryption randomness.[5] Thus, P_i^{hot}'s secret share 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. We will discuss how to generate the hot shares Section 6.6.

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 6.3.1) 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, our 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 7 in Section 6.3.1, the output of $\mathcal{H}$ is statistically indistinguishable from uniform.

[5] This means we can leave out the redundant first element which now equals vk, and each ciphertext will consist of a single group element.

6.5 Proofs of Remembrance

We will use non-replayable zero-knowledge proofs of knowledge (ZKPoKs) for two languages. We refer the reader to [Tha23b] 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 party 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 parties is more complicated. At setup time, each hot parties 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 $\mathcal{H}(\mathsf{ek}^x_{i,1}, \mathsf{ek}^x_{i,2}) + x_i$ 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 shares are unencrypted and then show how to adapt the construction to encrypted shares.

6.5.1 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 party 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-$(t-1)$ 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 $\mathsf{com_{ped}} := g_1^{f_T(i)} h_1^r$ and computes $\pi_{\mathsf{ped}} \leftarrow \Pi_{\mathsf{ped}}.\mathsf{Prove}(\mathsf{com_{ped}}; (f_T(i), r))$. 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 $(\mathsf{com_{ped}}, \pi_{\mathsf{ped}}, \overline{\pi}_{T,i}, g^{s_{T,i}(\tau)})$, where $s_{T,i}(X) := -r - s(X - i)$. The client accepts if and only if

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

(where the blue parts are changes to the original KZG verification check due to the blinding).

6.5.2 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 6.6). 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 party P_i^{hot} the opening $\tilde{f}_0(i) =: \tilde{x}_i$ and the corresponding evaluation proof $\pi_{0,i}$. Now the hot party can prove rembrance of its current share $\tilde{f}_T(i)$ in zero-knowledge in the same way as before, namely by committing to the share and blinding the evaluation proof $\pi_{T,i}$ as described in the previous section.

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

The final hot proof of remembrance is summarized as Π_{EKS} in Figure 6.3.

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 party 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 2.7).
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 $\pi_i^{\mathsf{hot}} := (\mathsf{com}_{\mathsf{ped}}, \pi_{\mathsf{ped}}, \overline{\pi}_{T,i}, S)$.

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

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

Figure 6.3: The proof system Π_{EKS} used by each P_i^{hot} to show possession of a valid encrypted key share with respect to the current KZG commitment.

6.6 Throback: Hot-Cold Threshold BLS with Proofs of Remembrance and Proactive Refresh

In this section, we show how to use the encryption scheme from Section 6.4 and the proofs of remembrance from Section 6.5 to construct a auditable hot-cold threshold backup using BLS signatures (recalled in Section 2.5.1).

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 (type-3) pairing between them. Since $\mathsf{vk} \in \mathbb{G}_2$, we will also instantiate the additive ElGamal encryption 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 2.5.1 and Section 6.4, respectively.

6.6.1 Subprotocols

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

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

– On input $(\texttt{sid}, \mathsf{PKSetup}, P)$, 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 $(P, \mathsf{ek}, \mathsf{dk}, \texttt{time} := 1, \texttt{unleaked} := 1)$ to L.
3. Output $(\texttt{sid}, \mathsf{PKSetupResult}, P, (\mathsf{ek}, \mathsf{dk}))$.

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

Figure 6.5: The public key functionality $\mathcal{F}_{\mathsf{PK}}$.

(Encrypted) secret share generation $\mathcal{F}_{\mathsf{SS}}$: Presented in Figure 6.4, this functionality is executed between a client C and a set of custodians (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 6.6.4, 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 6.5) which allows a party (institutional cold servers in our case) to obtain a secret (decryption) key dk sampled from $\mathbb{Z}_p$ while its public (encryption) key $\mathsf{ek} := g_2^{\mathsf{dk}}$ is made public, and can be retrieved reliably by any client. Again this functionality can be implemented by a party executing it locally. We abstract it out to separate the implementation details of this functionality from our modeling.

Construction Throback, our BLS-based auditable hot-cold threshold backup protocol, is given in Figures 6.6 and 6.7. Each cold storage is set up separately and independently of any client (user) via 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

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 2.5.1 and 6.4, resp.

SETUP

Registering a cold party 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 $t \leq n$ a signing threshold:

1. Call $\mathcal{F}_{\mathsf{PK}}$ on input $(\mathtt{sid}, \mathsf{PKRecover}, C, (P_i^{\mathsf{cold}}))$ for all $i \in [n]$. Waits for a response $(\mathtt{sid}, \mathsf{PKRecoverResult}, C, (P_i^{\mathsf{cold}}, \mathsf{ek}_i))$ for all $i \in [n]$.
2. Call $\mathcal{F}_{\mathsf{SS}}$ 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: To sign a message, a client C sends a a signature request $(\mathtt{sid}, \mathsf{TSign}, P_i, (C, \mathsf{vk}, m))$ to all $P_i \in \{P_1^{\mathsf{hot}}, P_1^{\mathsf{cold}}, \ldots, P_n^{\mathsf{hot}}, P_n^{\mathsf{cold}}\}$.

1. If P_i^{cold} decides to honor the request, it sends $\sigma_i^{\mathsf{cold}} := H(m)^r$ to P_i^{hot}, where $r := \mathcal{H}(\mathsf{vk}^{\mathsf{dk}_{i,1}}, \mathsf{vk}^{\mathsf{dk}_{i,2}})$.
2. If P_i^{hot} decides to honor the request, it retrieves $(C, \tilde{x}_i, *) \in L_i$ and computes $\sigma_i^{\mathsf{hot}} := H(m)^{\tilde{x}_i}$. Once it receives σ_i^{cold}, P_i^{hot} outputs $(\mathtt{sid}, \mathsf{TSignResult}, P_i^{\mathsf{hot}}, (C, m, \sigma_i := \sigma_i^{\mathsf{hot}} / \sigma_i^{\mathsf{cold}}))$.

Figure 6.6: The Throback protocol (setup and threshold signing).

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) by sending 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 $b_k \leftarrow \Pi_{\mathsf{DL}}.\mathsf{Verify}(\mathsf{ek}_{i,k}, \pi_{i,k}^{\mathsf{cold}})$ 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 $\tilde{x}_i$) by sending 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 $b \leftarrow \Pi_{\mathsf{EKS}}.\mathsf{Verify}((\mathsf{crs}, \mathsf{com}, i), \pi_i^{\mathsf{hot}})$ and outputs $(\mathtt{sid}, \mathsf{HProofResult}, C, (P_i^{\mathsf{hot}}, b))$.

Proactive Refresh

Hot share refresh: To trigger a refresh of its hot key shares, a client 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 6.7: The Throback protocol (proofs of remembrance and proactive refresh).

hot party also receives a proof π_i which will allow it to prove that $\tilde{x}_i$ was the output of $\mathsf{ClientRegister}$.

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,[6] each component can honor the request by producing a partial signature σ_i^{hot} or σ_i^{cold}, respectively, which the hot party combines 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 with which C registers 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 for 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 $\Pi_{\mathsf{EKS}}.\mathsf{Verify}(\mathsf{crs}, \mathsf{com}_T, \pi_i^{\mathsf{hot}}) = 1$ and $\Pi_{\mathsf{DL}}.\mathsf{Verify}((\mathsf{ek}_i, g_2), \pi_i^{\mathsf{cold}}) = 1$ for all $i \in S$, then $\mathsf{BLS}.\mathsf{Verify}(\mathsf{vk}, m, \mathsf{BLS}.\mathsf{Reconstruct}(\{\sigma_i\}_{i\in S})) = 1$ 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 Π_{DL} and Π_{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 parties. Each P_i^{cold} only stores a single decryption key $\mathsf{dk}_i \in \mathbb{G}_2^2$, regardless of the number of clients registered with its institution. To produce a cold partial signature, it computes two $\mathbb{G}_2$ exponentiations, one addition in $\mathbb{Z}_p$, and a single evaluation of $\mathcal{H}$, which is highly efficient since it is a simple subset sum (see Section 6.4). 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, a cold party 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.

[6] In most implementations, requests would be passed to each cold party via the corresponding hot party. To prevent a malicious hot party from sending spurious requests to its cold party, one could instantiate an authenticated channel over the hot parties between the client and each cold party.

6.6.2 Security Analysis

We prove our scheme secure in the universal composability framework[7], which we summarized in Section 2.8. Messages to and from the ideal functionalities consist of two pieces: a *header* and the *contents*. The header normally consists of a session identifier $\texttt{sid}$, a description of the action the functionality should take/has taken, and the sender/recipient. In this dissertation, 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 BLS auditable hot-cold threshold backup functionality $\mathcal{F}_{\mathsf{HC}}$, whose interfaces can only be called by *either* an institutional hot or cold party (specified by the superscripts hot and cold) or a client. For readability, we split $\mathcal{F}_{\mathsf{HC}}$ into four figures. The first, Figure 6.8, describes how parties register in the system. The share refresh and proof of remembrance interfaces of $\mathcal{F}_{\mathsf{HC}}$ are described in Figure 6.9. Signing is given in Figure 6.10 The adversarial interfaces (Figure 6.11) are described below. $\mathcal{F}_{\mathsf{HC}}$ uses the following internal variables to track the state of the system: $L_{\mathsf{cold}} = \{(P^{\mathsf{cold}}, \mathsf{ek}, \mathsf{dk}, \texttt{leaked}, \texttt{tampered}, \texttt{corrupted}, \texttt{allowc})\}$, 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; $L_{\mathsf{hot}} = \{(P^{\mathsf{hot}}, C_j, \tilde{x}, \pi, \texttt{time}, \texttt{leaked}, \texttt{tampered})\}$, which keeps track of the hot key material and whether it has been leaked or tampered with; and $S_{\mathsf{hot}} = \{P^{\mathsf{hot}}, \texttt{corrupted}, \texttt{allowc}\}$, which keeps track of hot corruptions.

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, i.e., 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.

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.

[7]Specifically, we use the version presented in [CLOS02a].

[8]We assume share refreshes are sequential and delivery to all P_i precedes any new refreshes.

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 $t \leq n$ a signing threshold, it proceeds as follows:

1. For each $i \in [n]$, retrieve $(P_i^{\mathsf{cold}}, \mathsf{ek}_i, *, *, *, *, *)$ from L_{cold}. If a public key for some party is unavailable then output $y := \perp$.
2. Otherwise, sample $x \leftarrow\!\!\$\, \mathbb{Z}_p \setminus \{0\}$. Let $y := g_2^x$.
3. Generate t-of-n Shamir shares of x as $x_1, \ldots x_n$. 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 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 6.8: The BLS auditable hot-cold threshold backup functionality $\mathcal{F}_{\mathsf{HC}}$ (registration).

Proactive Refresh

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

1. Output $\perp$ if a prior share refresh from C is still pending.[8]
2. Retrieve $(C, \mathcal{I}, t, y, \mathsf{com}, \texttt{time}) \in D$ for the maximum value of $\texttt{time}$.
3. Generate t-out-of-n Shamir shares of 0 as $x_1, \ldots, x_n \in \mathbb{Z}_p$ using polynomial f.
4. Compute $\mathsf{ucom} \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, f)$ and $\mathsf{com}' := \mathsf{com} \cdot \mathsf{ucom}$.
5. Add $(C, \mathcal{I}, t, y, \mathsf{com}', \texttt{time++})$ to D and send $(\texttt{sid}, \mathsf{ShareRefreshResult}, C, (1))$ to C.
6. For each $i \in [n]$:
   (a) Compute $(\delta_i, \pi'_i) \leftarrow \mathsf{KZG.Open}(\mathsf{crs}, f, i)$.
   (b) Send $(\texttt{sid}, \mathsf{ShareRefreshResult}, P_i^{\mathsf{hot}}, (C, b_i = 1))$ to P_i^{hot}.
   (c) Once $\mathcal{A}$ has allowed delivery of the message, retrieve $(P_i^{\mathsf{hot}}, C, \tilde{x}_i, \pi_i, T, *, \texttt{tampered}) \in L_{\mathsf{hot}}$ and add $(P_i^{\mathsf{hot}}, C, \tilde{x}_i + \delta_i, \pi_i \cdot \pi'_i, T\texttt{++}, \texttt{leaked} = 0, \texttt{tampered})$ to L_{hot}. Also retrieve $(P_i^{\mathsf{hot}}, *, \texttt{allowc}) \in S_{\mathsf{hot}}$ and update $\texttt{allowc}$ to 1.

Proofs of Remembrance

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

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

Figure 6.9: The BLS auditable hot-cold threshold backup functionality $\mathcal{F}_{\mathsf{HC}}$ (proactive refresh and proofs of remembrance).

Signing

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

Figure 6.10: The BLS auditable hot-cold threshold backup functionality $\mathcal{F}_{\mathsf{HC}}$ (signing).

Adversarial interference Figure 6.11 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 parties 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 (we weaken this assumption in Section 6.6.4).

Theorem 8 (security of Throback)**.** *Throback (Figures 6.6 and 6.7) 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 Section 6.6. 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.
- 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.

Leak

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

Tamper

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

Corrupt

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

Figure 6.11: The BLS auditable hot-cold threshold backup functionality $\mathcal{F}_{\mathsf{HC}}$ (adversarial interfaces).

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 $\mathsf{PVerify}(\mathsf{vk}_i, m, \sigma_i)$ 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 Section 2.5.1). 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 $b := \mathsf{PVerify}(\mathsf{vk}_i, m, \sigma_i^{\mathsf{hot}}/\sigma_i^{\mathsf{cold}*})$, 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 $b := \mathsf{PVerify}(\mathsf{vk}_i, m, \sigma_i^{\mathsf{hot}*}/\sigma_i^{\mathsf{cold}})$. 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 $\mathsf{PVerify}(\mathsf{vk}_i, m, \sigma_i^{\mathsf{hot}*}/\sigma_i^{\mathsf{cold}*}) = 1$ 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.

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

(a) Runtimes. The ShareRefresh time for P_i^{hot} includes running $\Pi_{\mathsf{Ref}}.\mathsf{HVerify}$ for the refresh proofs.

	Proof size (B)
Cold proof (Π_{DL})	256
Hot proof (Π_{EKS})	304
Refresh proof (ζ_i)	48

(b) Proof sizes

Table 6.1: Throback benchmarks independent of threshold

– 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 $b^* := \Pi_{\mathsf{DL}}.\mathsf{Verify}((\mathsf{ek}_i, g_2), \pi_i^{\mathsf{cold}})$. 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 $b^* := \Pi_{\mathsf{EKS}}.\mathsf{Verify}(\mathsf{com}, \pi_i^{\mathsf{hot}})$, 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}. □

From $\mathcal{F}_{\mathsf{HC}}$ to threshold BLS 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 7.

	Time (ms)		
$(t, n) =$	$(3, 5)$	$(5, 20)$	$(67, 100)$
$\mathsf{ClientRegister}$	10	40	170
$\mathsf{ShareRefresh}$ (C)	11	41	172
Hot Prove ($\Pi_{\mathsf{EKS}}.\mathsf{Prove}$)	10	40	170
Hot Verify ($\Pi_{\mathsf{EKS}}.\mathsf{Verify}$)	14	43	194

Table 6.2: Throback benchmarks for each setting of (t, n). The $\mathsf{ShareRefresh}$ times for C includes running $\Pi_{\mathsf{Ref}}.\mathsf{UCVerify}$.

6.6.3 Implementation and Evaluation

We implemented our construction in Rust using the BLS12-381 elliptic curve.[9] Each element in $\mathbb{G}_1$ is 48 bytes in compressed form, $\mathbb{G}_2$ is 96 bytes, and $\mathbb{Z}_p$ is 32 bytes. Thus the size of vk is 96 bytes. Each ek_i is 192 bytes (2 $\mathbb{G}_2$ elements). The hot shares $\tilde{x}_i$ and share refresh information δ_i are 32 bytes, and the cold shares dk_i are 64 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 $\Pi_{\mathsf{Ref}}.\mathsf{HVerify}$ to check ζ_i) are all independent of (t, n) and are shown in Table 6.1a. Similarly, proof sizes are independent of (t, n) and are given in Table 6.1b. Runtimes for generating hot shares ($\mathsf{ClientRegister}$), hot share refreshes (using the realization of $\mathcal{F}_{\mathsf{SS}}$ shown in Figure 6.14), and hot proofs depend on the specific values of (t, n) and are shown in Table 6.2. The benchmarks are an average over 1000 iterations using a machine with an 8-core AMD Ryzen 9 5900HX processor at 5GHz with 64GB RAM and 512KB L1, 4MB L2, and 16 MB L3 cache.

6.6.4 Trustless Proactive Refresh Using a Bulletin-Board

Recall from Section 6.5.2 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 $X = 0$.

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

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 $\mathsf{KZG.Verify}(\mathsf{crs}, \mathsf{ucom}_T, i, \delta_i, \zeta_i) = 1$, and only update their key share and opening proof if verification passes.

SHARE REFRESH PROOFS (Π_{Ref})

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

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

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

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

$\mathsf{UCVerify}((\mathsf{crs}, \mathsf{ucom}_T, t-1), \pi_{\mathsf{ucom}}) \to \{0,1\}$: 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:

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

Figure 6.12: 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 $\mathsf{HVerify}$, and the correctness of ucom is verified separately via $\mathsf{UCVerify}$.

Figure 6.12 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 $\mathsf{UCVerify}$) and of each

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

- On input $(\mathtt{sid}, \mathsf{Setup}, C, (\mathsf{vk}, t, \mathsf{com}))$, delete any existing entries $(C, *, *, *, *) \in D$ and $(C, *, *) \in U$. Add $(C, \mathsf{vk}, t, \mathsf{com}, \mathtt{time} := 1)$ 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 $\Pi_{\mathsf{Ref}}.\mathsf{UCVerify}((\mathsf{crs}, \mathsf{ucom}, t), \pi_{\mathsf{ucom}}) = 1$, 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 $(\mathtt{sid}, \mathsf{AuxInfo}, P, (C, \mathsf{vk}, \mathsf{com}))$.
- 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 6.13: The programmable bulletin-board functionality $\mathcal{F}_{\mathsf{BB}}$

hot party's refresh information δ_i (via $\mathsf{HVerify}$).

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 6.13). 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 updating the stored commitment. Additionally, C will let $\mathcal{F}_{\mathsf{BB}}$ store and distribute ucom instead. We describe the full protocol in Figure 6.14.

Theorem 9. *The encrypted secret sharing protocol in Figure 6.14 UC-realizes $\mathcal{F}_{\mathsf{SS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model.*

Proof. Let $\mathcal{A}$ be the adversary interacting with the parties $P_1, \ldots, P_n, C$ running the protocol in Figure 6.14. We will construct a simulator $\mathcal{S}$ running in the ideal world so that no environment $\mathcal{E}$ can distinguish an execution of the ideal-world interaction from the real protocol. $\mathcal{S}$ will interact with $\mathcal{F}_{\mathsf{SS}}$, $\mathcal{E}$, and invoke a copy of $\mathcal{A}$ to run a simulated interaction of the protocol (which we again refer

Encrypted secret sharing protocol

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

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

1. Sample $x \leftarrow\!\!\$\ \mathbb{Z}_p \setminus \{0\}$. 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 $t \leq n \leq d$, and $\mathcal{P} = \{P_1 \dots P_n\}$ a set of parties, C will proceed as follows:

1. Generate t-out-of-n Shamir Shares of 0 as $x_1, \dots x_n \in \mathbb{Z}_p$; let f be the polynomial used.
2. Compute $\mathsf{com}_0 \leftarrow \mathsf{KZG.Com}(\mathsf{crs}, f)$ and $(\{\zeta_i\}_{i\in[n]}, \pi_z) \leftarrow \Pi_{\mathsf{Ref}}.\mathsf{Prove}((\mathsf{crs}, \mathsf{com}_0, t-1, \{x_i\}_{i\in[n]}); f(X))$. Send $(\mathtt{sid}, \mathsf{ComUpdate}, C, (\mathsf{com}_0, \pi_z))$ to $\mathcal{F}_{\mathsf{BB}}$, which returns $(\mathtt{sid}, \mathsf{ComUpdateResult}, C, (b))$.
3. Send (x_i, ζ_i) to P_i for all $i \in [n]$, then output $(\mathtt{sid}, \mathsf{ZeroSetupDone}, C, (b))$.
4. Each party P_i for $i \in [n]$ will send $(\mathtt{sid}, \mathsf{UComRecover}, P_i, (C))$ to $\mathcal{F}_{\mathsf{BB}}$ and receive $(\mathtt{sid}, \mathsf{UCom}, P_i, (C, \mathsf{ucom}))$ in return. It checks that $\Pi_{\mathsf{Ref}}.\mathsf{HVerify}((\mathsf{crs}, \mathsf{ucom}, i, x_i), \zeta_i) = 1$. 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 $(\mathtt{sid}, \mathsf{ClientInfoRecover}, P, (C))$ to $\mathcal{F}_{\mathsf{BB}}$.

Figure 6.14: Protocol realizing $\mathcal{F}_{\mathsf{SS}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model. Changes with respect to locally running the ideal functionality $\mathcal{F}_{\mathsf{SS}}$ are shown in blue.

to as the internal interaction). The dummy parties, communication tapes, and message delivery are handled in the same way as in the proof of Theorem 8.

Corruptions Whenever $\mathcal{A}$ corrupts a party P_i in the internal execution, $\mathcal{S}$ corrupts the corresponding dummy party.

Setup and auxiliary information Since all the computation in setup is done by C and we do not allow it to be corrupted in this phase, $\mathcal{S}$ only passes messages to and from $\mathcal{F}_{\mathsf{SS}}$. It also keeps track of client-threshold pairs in a list D'. Similarly, retrieving auxiliary information involves no computation, so $\mathcal{S}$ again only passes along messages.

Generating share refreshes If C is corrupt, $\mathcal{S}$ retrieves $(C, t) \in D'$ and observes what values $\mathsf{ucom}, \pi_{\mathsf{ucom}}$ the adversary $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{BB}}$ in the internal interaction. It computes $b^* \leftarrow \Pi_{\mathsf{Ref}}.\mathsf{UCVerify}((\mathsf{crs}, \mathsf{ucom}, t), \pi_{\mathsf{ucom}})$. When $\mathcal{F}_{\mathsf{SS}}$ sends $\mathcal{S}$ a $\mathsf{ZeroSetupRequest}$ message, it responds with b^*.

$\mathcal{S}$ also keeps track of the values (x_i, ζ_i) sent by $\mathcal{A}$ to each P_i in the internal interaction and computes $b_i^* \leftarrow \Pi_{\mathsf{Ref}}.\mathsf{HVerify}((\mathsf{crs}, \mathsf{ucom}, i, x_i), \zeta_i)$. When $\mathcal{F}_{\mathsf{SS}}$ sends $\mathcal{S}$ a $\mathsf{ZeroShareRequest}$ message, it responds with b_i^*. $\square$

Chapter 7

Conclusion

In this dissertation, I gave four ways in which advanced cryptography can improve the security and trust assumptions of blockchain applications in a practical and deployable manner: naysayer proofs, the formalization of blind conditional signatures, the Cicada framework for fair and non-interactive voting and auctions, and Throback, a BLS-based auditable hot-cold threshold backup. Given that 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 them as a building block. A more theoretical analysis of naysayer complexity, lower bounds, and application to particular classes of underlying verifiers or popular proof systems may also be fruitful. Finally, like optimistic rollups, naysayer proofs would benefit from a rigorous analysis of how to set collateral and delay periods in optimistic settings.

Second, the BCS primitive can be used to analyze any coin mixing protocol which falls into the synchronization puzzle paradigm. Extensions to this definition can broaden its applicability. Designing a more efficient universally-composable BCS construction is another direction which can be of interest in practical deployments, where a protocol is part of a large, complex system and thus security should be analyzed rigorously and ideally hold under composition.

Next, I showed that the Cicada framework is a useful blueprint for fair on-chain elections and auctions and demonstrated its practicality for many popular protocols. Extending Cicada to other, non-additive scoring protocols could be a useful future work. Each of the proposed extensions can be explored in greater detail as well.

Finally, the auditable hot-cold threshold backup protocol Throback is designed for a popular threshold signature scheme used in the blockchain ecosystem today. It may be desirable or necessary to extend these ideas to other threshold signatures (e.g., pairing-free signatures like Schnorr). Furthermore, although Throback already offers several advanced functionalities, deployments may wish to extend it even 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, I expect that cryptographic protocols and primitives will continue to play a large role in furthering security, decentralization, and scalability.

Bibliography

[AAB+24] Mario M Alvarez, Henry Arneson, Ben Berger, Lee Bousfield, Chris Buckland, Yafah Edelman, Edward W Felten, Daniel Goldman, Raul Jordan, Mahimna Kelkar, et al. Bold: Fast and cheap dispute resolution. *arXiv preprint arXiv:2404.10491*, 2024. *[p. 14]*

[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. 85]*

[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. 24]*

[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. *[p. 106]*

[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. *[pp. 25, 72, 73, and 76]*

[AEE+21a] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized channels from limited blockchain scripts and adaptor signatures. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 635–664. Springer, Cham, December 2021. *[p. 39]*

[AEE+21b] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized channels from limited blockchain scripts and adaptor signatures. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part II 27*, pages 635–664. Springer, 2021. *[p. 68]*

[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. 82]*

[AHS20] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. Cryptology ePrint Archive, Report 2020/1390, 2020. *[p. 110]*

[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, Cham, May 2019. *[pp. 114 and 115]*

[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. *[p. 110]*

[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, Berlin, Heidelberg, August 2015. *[pp. 72 and 73]*

[Arm16] Brian Armstrong. How coinbase builds secure infrastructure to store bitcoin in the cloud. `https://medium.com/the-coinbase-blog/how-coinbase-builds-secure-infrastructure-to-store-bitcoin-in-the-cloud-30a6504e40ba`, apr 2016. *[p. 107]*

[Azt] Aztec documentation. `https://docs.aztec.network/`. *[p. 12]*

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Cham, August 2018. *[p. 74]*

[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, Cham, August 2023. *[p. 73]*

[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, Cham, August 2019. *[pp. 90, 92, 93, and 103]*

[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. 23]*

[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. *[p. 23]*

[BBSU12] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better - how to make Bitcoin a better currency. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 399–414. Springer, Berlin, Heidelberg, February / March 2012. *[p. 32]*

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Berlin, Heidelberg, February 2009. *[pp. 72 and 73]*

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. *[p. 38]*

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 315–333. Springer, Berlin, Heidelberg, March 2013. *[p. 67]*

[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, Berlin, Heidelberg, February 2010. *[p. 89]*

[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, Cham, August 2022. *[p. 110]*

[BD21] Jeffrey Burdges and Luca De Feo. Delay encryption. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 302–326. Springer, Cham, October 2021. *[p. 74]*

[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, Berlin, Heidelberg, February / March 2006. *[p. 73]*

[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. 82]*

[BEHG20] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Advances in Financial Technologies*, 2020. *[p. 25]*

[BF01] Dan Boneh and Matthew Franklin. Efficient generation of shared RSA keys. *Journal of the ACM (JACM)*, 48(4):702–722, 2001. *[p. 83]*

[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. 27]*

[BFP21] Balthazar Bauer, Georg Fuchsbauer, and Antoine Plouviez. The one-more discrete logarithm assumption in the generic group model. Cryptology ePrint Archive, Report 2021/866, 2021. *[p. 41]*

[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, Berlin, Heidelberg, April 2012. *[p. 25]*

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision Diffie-Hellman. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 108–125. Springer, Berlin, Heidelberg, August 2008. *[p. 46]*

[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. 24 and 25]*

[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. 96]*

[BHSR19] Samiran Bag, Feng Hao, Siamak F Shahandashti, and Indranil Ghosh Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15, 2019. *[p. 74]*

[BLM22] Michele Battagliola, Riccardo Longo, and Alessio Meneghetti. Extensible decentralized secret sharing and application to schnorr signatures. Cryptology ePrint Archive, Report 2022/1551, 2022. *[p. 110]*

[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, Berlin, Heidelberg, December 2001. *[pp. 6, 7, and 74]*

[BMP22] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. Efficient asymmetric threshold ECDSA for MPC-based cold storage. Cryptology ePrint Archive, Report 2022/1296, 2022. *[p. 114]*

[BN00] Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254. Springer, Berlin, Heidelberg, August 2000. *[p. 74]*

[BNM$^+$14] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 486–504. Springer, Berlin, Heidelberg, March 2014. *[p. 32]*

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003. *[p. 41]*

[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, Berlin, Heidelberg, January 2003. *[p. 62]*

[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. 76]*

[Buc] Chris Buckland. Fraud proofs and virtual machines. `https://medium.com/@cpbuckland88/fraud-proofs-and-virtual-machines-2826a3412099`. *[p. 14]*

[But13] Vitalik Buterin. Deterministic wallets, their advantages and their understated flaws. `https://bitcoinmagazine.com/technical/deterministic-wallets-advantages-flaw-1385450276`, 11 2013. *[p. 106]*

[But14] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *White paper*, 2014. *[p. 1]*

[Cac99] Christian Cachin. Efficient private bidding and auctions with an oblivious third party. In Juzar Motiwalla and Gene Tsudik, editors, *ACM CCS 99*, pages 120–127. ACM Press, November 1999. *[p. 73]*

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5), September 2020. *[pp. 8 and 32]*

[CATB23] Kevin Choi, Arasu Arun, Nirvan Tyagi, and Joseph Bonneau. Bicorn: An optimistically efficient distributed randomness beacon. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023, Part I*, volume 13950 of *LNCS*, pages 235–251. Springer, Cham, May 2023. *[p. 74]*

[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, Cham, May 2020. *[p. 110]*

[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. *[p. 110]*

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Berlin, Heidelberg, February 2007. *[p. 59]*

[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, Berlin, Heidelberg, August 1994. *[pp. 90 and 97]*

[CEG+21] Mihai Christodorescu, Erin English, Wanyun Catherine Gu, David Kreissman, Ranjit Kumaresan, Mohsen Minaei, Srinivasan Raghuraman, Cuy Sheffield, Arjuna Wijeyekoon, and Mahdi Zamani. Universal payment channels: An interoperability platform for digital currencies. *arXiv preprint arXiv:2109.12194*, 2021. *[p. 32]*

[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, Berlin, Heidelberg, February / March 2013. *[p. 27]*

[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. *[p. 110]*

[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, Cham, 2016. *[pp. 72 and 73]*

[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, Berlin, Heidelberg, May 1997. *[p. 80]*

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), 1981. *[p. 25]*

[Cha82] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982. *[pp. 33, 37, and 51]*

[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. 83]*

[Chi24] Chia Network. BLS keys. `https://docs.chia.net/bls-keys/`, 2024. *[p. 110]*

[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, Cham, May 2020. *[p. 134]*

[CJSS21] Peter Chvojka, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Versatile and sustainable timed-release encryption and sequential timelock puzzles (extended abstract). In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021, Part II*, volume 12973 of *LNCS*, pages 64–85. Springer, Cham, October 2021. *[pp. 73, 74, 103, and 104]*

[CKS11] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 449–467. Springer, Berlin, Heidelberg, December 2011. *[p. 38]*

[CLOS02a] 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. *[pp. 9 and 126]*

[CLOS02b] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002. *[p. 59]*

[Coi] Coinbase. What does coinbase do with my digital assets? `https://help.coinbase.com/en/coinbase/other-topics/legal-policies/what-does-coinbase-do-with-my-digital-assets`. *[p. 107]*

[Coi22] Coinswap, Accessed on May 2022. `https://coinswap.space`. *[p. 32]*

[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, Berlin, Heidelberg, August 1993. *[pp. 26, 90, and 97]*

[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, Berlin, Heidelberg, August 1998. *[pp. 69 and 103]*

[DEFM19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*, pages 106–123. IEEE Computer Society Press, May 2019. *[p. 38]*

[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. *[p. 106]*

[DHMW23] Nico Döttling, Lucjan Hanzlik, Bernardo Magri, and Stella Wohnig. McFly: Verifiable encryption to the future made practical. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023, Part I*, volume 13950 of *LNCS*, pages 252–269. Springer, Cham, May 2023. *[p. 74]*

[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. *[p. 110]*

[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, Berlin, Heidelberg, September 2013. *[p. 102]*

[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. *[p. 24]*

[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. 24]*

[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. *[p. 110]*

[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. 72 and 73]*

[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, Berlin, Heidelberg, February 2010. *[p. 83]*

[DMP88] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge proof systems. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 52–72. Springer, Berlin, Heidelberg, August 1988. *[p. 38]*

[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. 102 and 103]*

[dYd] dYdX. `https://dydx.exchange/`. *[p. 12]*

[Edg23] Ben Edgington. *Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond.* n.p., 2023. `https://eth2book.info/latest/`. *[p. 110]*

[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, Berlin, Heidelberg, February 2004. *[p. 72]*

[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, Berlin, Heidelberg, August 1984. *[pp. 112 and 116]*

[Eme13] Peter Emerson. The original Borda count and partial voting. *Social Choice and Welfare*, 40:353–358, 2013. *[pp. 77 and 86]*

[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, Cham, September 2022. *[p. 106]*

[Eth19] Ethereum Foundation. Semaphore: a zero-knowledge set implementation for ethereum. `https://github.com/semaphore-protocol/semaphore`, May 2019. *[p. 76]*

[Eth23a] Ethereum. Account Abstraction. `ethereum.org/en/roadmap/account-abstraction`, 2023. *[p. 24]*

[Eth23b] Ethereum. Optimistic rollups. `https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/`, 2023. *[p. 14]*

[Eth24a] Ethereum. Account abstraction. `https://ethereum.org/en/roadmap/account-abstraction/`, march 2024. *[p. 110]*

[Eth24b] Ethereum. Merkle patricia trie. `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/`, jul 2024. *[p. 22]*

[Eur21] European CentralBank. Digital euro, August 2021. `https://www.ecb.europa.eu/paym/digital_euro/html/index.en.html`. *[p. 32]*

[FFH+24] Rainer Feichtinger, Robin Fritsch, Lioba Heimbach, Yann Vonlanthen, and Roger Wattenhofer. Sok: Attacks on daos. *arXiv preprint arXiv:2406.15071*, 2024. *[p. 72]*

[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. 77]*

[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. 24]*

[Fic] Kelvin Fichter. Why is the optimistic rollup challenge period 7 days? `https://kelvinfichter.com/pages/thoughts/challenge-periods/`. *[p. 14]*

[Fil] Filecoin. Filecoin spec. `https://spec.filecoin.io/algorithms/crypto/signatures/`. *[p. 110]*

[FK97] Uriel Feige and Joe Kilian. Making games short (extended abstract). In *29th ACM STOC*, pages 506–516. ACM Press, May 1997. *[p. 14]*

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Cham, August 2018. *[p. 34]*

[FKPS21] Cody Freitag, Ilan Komargodski, Rafael Pass, and Naomi Sirkin. Non-malleable time-lock puzzles and applications. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 447–479. Springer, Cham, November 2021. *[pp. 74 and 81]*

[FMW22] Robin Fritsch, Marino Müller, and Roger Wattenhofer. Analyzing voting power in decentralized governance: Who controls DAOs? *arXiv preprint arXiv:2204.01176*, 2022. *[p. 72]*

[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. *[pp. 72 and 73]*

[FR96] Matthew K Franklin and Michael K Reiter. The design and implementation of a secure auction service. *IEEE Transactions on Software Engineering*, 22(5):302–312, 1996. *[p. 73]*

[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, Berlin, Heidelberg, August 1987. *[pp. 8, 92, and 118]*

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009. *[pp. 49 and 72]*

[GFW22] Simin Ghesmati, Walid Fdhila, and Edgar Weippl. Sok: How private is bitcoin? classification and evaluation of bitcoin privacy techniques. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery. *[p. 32]*

[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. *[p. 110]*

[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`. *[pp. 72 and 73]*

[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. *[p. 110]*

[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, Berlin, Heidelberg, August 2010. *[p. 11]*

[Gir20] Damien Giry. Cryptographic key length recommendation. `https://www.keylength.com/`, May 2020. *[p. 24]*

[GKSŚ20] Adam Gągol, Jędrzej Kula, Damian Straszak, and Michał Świętek. Threshold ECDSA for decentralized asset custody. Cryptology ePrint Archive, Report 2020/498, 2020. *[p. 110]*

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th ACM STOC*, pages 365–377. ACM Press, May 1982. *[pp. 41 and 60]*

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988. *[p. 38]*

[GMR23] Nicolas Gailly, Kelsey Melissaris, and Yolan Romailler. tlock: Practical timelock encryption from threshold BLS. Cryptology ePrint Archive, Report 2023/189, 2023. *[p. 74]*

[Gol01] Oded Goldreich. *Foundations of Cryptography*, volume 1. Cambridge University Press, 2001. *[p. 4]*

[Gro04] Jens Groth. Rerandomizable and replayable adaptive chosen ciphertext attack secure cryptosystems. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 152–170. Springer, Berlin, Heidelberg, February 2004. *[pp. 33, 37, and 41]*

[Gro05] Jens Groth. Non-interactive zero-knowledge arguments for voting. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05International Conference on Applied Cryptography and Network Security*, volume 3531 of *LNCS*, pages 467–482. Springer, Berlin, Heidelberg, June 2005. *[pp. 73, 81, 90, 94, 97, and 99]*

[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, Berlin, Heidelberg, May 2016. *[pp. 19 and 100]*

[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. 82]*

[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. *[pp. 21 and 100]*

[GY18] Hisham S. Galal and Amr M. Youssef. Verifiable sealed-bid auction on the Ethereum blockchain. Cryptology ePrint Archive, Report 2018/704, 2018. *[p. 74]*

[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. *[pp. 72 and 73]*

[HAB+17] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS 2017*. The Internet Society, February / March 2017. *[pp. 32, 33, 34, and 35]*

[Hab22] Ulrich Haböck. A summary on the FRI low degree test. Cryptology ePrint Archive, Report 2022/1216, 2022. *[p. 24]*

[HBG16] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 43–60. Springer, Berlin, Heidelberg, February 2016. *[p. 32]*

[HK07] Omer Horvitz and Jonathan Katz. Universally-composable two-party computation in two rounds. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 111–129. Springer, Berlin, Heidelberg, August 2007. *[p. 59]*

[HMMP23] Thomas Haines, Rafieh Mosaheb, Johannes Müller, and Ivan Pryvalov. SoK: Secure E-voting with everlasting privacy. *PoPETs*, 2023(1):279–293, January 2023. *[p. 73]*

[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, Berlin, Heidelberg, May 2000. *[p. 81]*

[HSRK21] Bernhard Haslhofer, Rainer Stütz, Matteo Romiti, and Ross King. Graphsense: A general-purpose cryptoasset analytics platform. *arXiv preprint arXiv:2102.13613*, 2021. *[p. 32]*

[HTK98] Michael Harkavy, J Doug Tygar, and Hiroaki Kikuchi. Electronic auctions with private bids. In *USENIX Workshop on Electronic Commerce*, pages 61–74, 1998. *[p. 73]*

[Hu23] Mingxing Hu. Post-quantum secure deterministic wallet: Stateless, hot/cold setting, and more secure. Cryptology ePrint Archive, Report 2023/062, 2023. *[p. 106]*

[HZ06] Feng Hao and Piotr Zieliński. A 2-round anonymous veto protocol. In *Security Protocols Workshop*, 2006. *[p. 74]*

[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. 114 and 115]*

[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. 76]*

[Kat24] Jonathan Katz. Round-optimal, fully secure distributed key generation. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 285–316. Springer, Cham, August 2024. *[p. 114]*

[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. *[pp. 69 and 102]*

[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, Cham, October 2020. *[p. 110]*

[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. 14 and 17]*

[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. 100]*

[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. 82]*

[Kra] Kraken. Cryptocurrency security protocols. `https://www.kraken.com/features/security`. *[p. 107]*

[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, Berlin, Heidelberg, December 2010. *[pp. 7, 27, and 111]*

[Lab23] Offchain Labs. Inside Arbitrum Nitro. `https://docs.arbitrum.io/inside-arbitrum-nitro/`, 2023. *[p. 14]*

[Lit24] Lit Protocol. Lit protocol whitepaper v1. `https://github.com/LIT-Protocol/whitepaper`, 2024. *[p. 107]*

[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. 110 and 114]*

[LOS+06] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 465–485. Springer, Berlin, Heidelberg, May / June 2006. *[p. 62]*

[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. 76]*

[LRR+19] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. Omniring: Scaling private payments without trusted setup. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 31–48. ACM Press, November 2019. *[p. 38]*

[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. 77 and 92]*

[Mau05] Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Berlin, Heidelberg, December 2005. *[p. 34]*

[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, Berlin, Heidelberg, August 1988. *[pp. 22 and 27]*

[MLQ23] Liam Medley, Angelique Faye Loe, and Elizabeth A. Quaglia. SoK: Delay-based cryptography. In *CSF 2023 Computer Security Foundations Symposium*, pages 169–183. IEEE Computer Society Press, July 2023. *[p. 74]*

[MM18] Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless tumbling for transaction privacy. *PoPETs*, 2018(2):105–121, April 2018. *[p. 37]*

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Publicly verifiable proofs of sequential work. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 373–388. ACM, January 2013. *[p. 74]*

[MPJ+13] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140, 2013. *[p. 32]*

[MSH17] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In Aggelos Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 357–375. Springer, Cham, April 2017. *[p. 73]*

[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, Cham, August 2019. *[pp. 73, 74, 75, 78, 79, and 103]*

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008. *[p. 1]*

[Nat15] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015. `http://dx.doi.org/10.6028/NIST.FIPS.202`. *[p. 69]*

[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. 76]*

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 129–139, 1999. *[p. 73]*

[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 24International Conference on Applied Cryptography and Network Security, Part III*, volume 14585 of *LNCS*, pages 105–134. Springer, Cham, March 2024. *[p. 103]*

[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, Berlin, Heidelberg, August 1993. *[pp. 8 and 9]*

[Ope23] OpeanSea. How to sell an NFT, June 2023. *[p. 72]*

[OPP14] Rafail Ostrovsky, Anat Paskin-Cherniavsky, and Beni Paskin-Cherniavsky. Maliciously circuit-private FHE. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 536–553. Springer, Berlin, Heidelberg, August 2014. *[p. 43]*

[Opt23a] Optimism. Rollup Protocol. `https://community.optimism.io/docs/protocol/2-rollup-protocol`, 2023. *[p. 14]*

[Opt23b] Optimism. What is the Optimism Collective?, February 2023. *[p. 72]*

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Berlin, Heidelberg, May 1999. *[p. 78]*

[PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017. *[p. 17]*

[PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. `https://lightning.network/lightning-network-paper.pdf`, 2016. *[p. 17]*

[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, Berlin, Heidelberg, August 1992. *[pp. 8 and 89]*

[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, Cham, December 2021. *[p. 110]*

[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. 24]*

[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. *[p. 82]*

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019. *[p. 89]*

[Pol78] John M Pollard. Monte carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978. *[p. 95]*

[PR07] Manoj Prabhakaran and Mike Rosulek. Rerandomizable RCCA encryption. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 517–534. Springer, Berlin, Heidelberg, August 2007. *[p. 38]*

[PRF23] Mallesh Pai, Max Resnick, and Elijah Fox. Censorship resistance in on-chain auctions. *arXiv preprint arXiv:2301.13321*, 2023. *[p. 72]*

[Pri] Privacy and Scaling Explorations. Rate-limiting nullifier. `https://rate-limiting-nullifier.github.io/rln-docs/`. *[pp. 72, 73, and 76]*

[Pri22] Privacy & Scaling Explorations. BLS wallet: Bundling up data. `https://medium.com/privacy-scaling-explorations/bls-wallet-bundling-up-data-fb5424d3bdd3`, August 2022. *[p. 110]*

[Pri23] Privacy and Scaling Explorations. Maci: Minimal anti-collusion infrastructure. `https://maci.pse.dev/`, 2023. *[pp. 72 and 73]*

[PRST06] David C Parkes, Michael O Rabin, Stuart M Shieber, and Christopher A Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. In *Proceedings of the 8th international conference on Electronic commerce: The new e-commerce: innovations for conquering current barriers, obstacles and limitations to conducting successful business on the internet*, pages 70–81, 2006. *[p. 73]*

[PSS19] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4, 2019. *[p. 76]*

[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. 89 and 99]*

[RM17] Tim Ruffing and Pedro Moreno-Sanchez. ValueShuffle: Mixing confidential transactions for comprehensive transaction privacy in bitcoin. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *FC 2017 Workshops*, volume 10323 of *LNCS*, pages 133–154. Springer, Cham, April 2017. *[pp. 32 and 37]*

[RMK14] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. CoinShuffle: Practical decentralized coin mixing for bitcoin. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 345–364. Springer, Cham, September 2014. *[pp. 32 and 37]*

[RMK16] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2P mixing and unlinkable bitcoin transactions. Cryptology ePrint Archive, Report 2016/824, 2016. *[pp. 32 and 37]*

[Rob19] Daniel Robinson. HTLCs considered harmful, 2019. `https://cbr.stanford.edu/sbc19/`. *[p. 59]*

[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, Cham, November 2021. *[p. 100]*

[RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, 1996. *[pp. 74, 75, and 78]*

[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, Berlin, Heidelberg, November 1999. *[p. 83]*

[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, New York, August 1990. *[pp. 19, 90, 103, and 118]*

[Scr] Scroll. `https://scroll.io/`. *[p. 12]*

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979. *[pp. 5, 106, and 111]*

[SHMSM21] Johann Stockinger, Bernhard Haslhofer, Pedro Moreno-Sanchez, and Matteo Maffei. Pinpointing and measuring wasabi and samourai coinjoins in the bitcoin ecosystem. *arXiv preprint arXiv:2109.10229*, 2021. *[p. 32]*

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Berlin, Heidelberg, May 1997. *[p. 34]*

[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. 14]*

[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. *[p. 17]*

[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. *[pp. 17, 25, and 38]*

[Sta] Starknet. `https://www.starknet.io/`. *[p. 12]*

[Sto22] Jeff Stone. Evolution downfall: Insider 'exit scam' blamed for massive drug bazaar's sudden disappearance, Accessed on May 2022. `https://www.ibtimes.com/evolution-downfall-insider-exit-scam-blamed-massive-drug-bazaars-sudden-disappearance-1856190`. *[p. 32]*

[SU17] Dominique Schröder and Dominique Unruh. Security of blind signatures revisited. *Journal of Cryptology*, 30(2):470–494, April 2017. *[pp. 33 and 37]*

[SV05] Nigel Smart and Frederik Vercauteren. On computable isomorphisms in efficient asymmetric pairing based systems. Cryptology ePrint Archive, Report 2005/116, 2005. *[p. 6]*

[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, Berlin, Heidelberg, January 2003. *[p. 72]*

[TAF+23] Nirvan Tyagi, Arasu Arun, Cody Freitag, Riad Wahby, Joseph Bonneau, and David Mazières. Riggs: Decentralized sealed-bid auctions. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 1227–1241. ACM Press, November 2023. *[pp. 72, 74, and 87]*

[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. 82]*

[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. 83, 88, and 96]*

[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. 85]*

[Tha23a] Justin Thaler. 17 misconceptions about SNARKs (and why they hold us back). `https://a16zcrypto.com/posts/article/17-misconceptions-about-snarks`, Jul 2023. *[p. 4]*

[Tha23b] Justin Thaler. Proofs, arguments, and zero-knowledge. `https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf`, July 2023. *[pp. 4 and 118]*

[TMM21] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A^{2}L: Anonymous atomic locks for scalability in payment channel hubs. In *2021 IEEE Symposium on Security and Privacy*, pages 1834–1851. IEEE Computer Society Press, May 2021. *[pp. 32, 33, 34, 35, 37, 42, 43, 51, 59, and 60]*

[TR19] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *arXiv preprint arXiv:1908.04756*, 2019. *[p. 14]*

[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. 82]*

[VR15] Luke Valenta and Brendan Rowan. Blindcoin: Blinded, accountable mixes for bitcoin. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 112–126. Springer, Berlin, Heidelberg, January 2015. *[p. 32]*

[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, Cham, May 2019. *[pp. 89, 98, and 99]*

[Wik22] Bitcoin Wiki. CoinJoin, Accessed on May 2022. `https://en.bitcoin.it/wiki/CoinJoin`. *[pp. 32 and 37]*

[Woo24] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`, june 2024. *[p. 11]*

[Wui12] Pieter Wuille. BIP32: Hierarchical deterministic wallets. `https://en.bitcoin.it/wiki/BIP_0032`, February 2012. *[p. 106]*

[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. 72]*

[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. 111 and 119]*

[ZKs] ZKsync. `https://www.zksync.io/`. *[p. 12]*