ABSTRACT

Title of Dissertation: DATA STRUCTURES AND PROTOCOLS FOR SCALABILITY AND SECURITY OF DISTRIBUTED CONSENSUS

Shravan Srinivasan
Doctor of Philosophy, 2023

Dissertation Directed by: Professor Charalampos Papamanthou
Department of Computer Science

Distributed consensus is the problem of reaching an agreement among mutually distrusting nodes in the presence of faults. Despite tremendous progress in achieving internet-scale consensus, prior consensus protocols face scalability and security challenges. This dissertation proposes new authenticated data structures and protocols to improve the scalability (through *stateless* blockchains) and security (through support for more powerful network adversaries) of distributed consensus, respectively.

In the first part, we present our Vector Commitment (VC) data structure, Hyperproofs, to improve scalability of blockchains. Our VC is the first construction that is efficiently *maintainable* (can update all proofs in sublinear time) and *aggregatable* (can combine multiple individual proofs into a single succinct proof). Hyperproofs also incentivize proof computation through a new property called *unstealability*, which allows a prover to cryptographically bind the proofs she computes irreversibly with her identity. Finally, we

present schemes to succinctly prove and verify the (non-)membership of multiple elements in a cryptographic *accumulator* for applications in the distributed setting.

In the second part, we present protocols to improve security of distributed consensus against powerful network adversaries that can delay or delete any message in the following settings: (1) We describe the *first* protocol to show that it is possible to achieve permissionless consensus even after relaxing the standard synchrony assumption. (2) We present the *first* expected constant-round Byzantine broadcast under a *strongly adaptive* and *dishonest majority* setting.

DATA STRUCTURES AND PROTOCOLS FOR SCALABILITY AND SECURITY OF DISTRIBUTED CONSENSUS

by

Shravan Srinivasan

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
2023

Advisory Committee:
Professor Charalampos Papamanthou, Chair/Advisor
Professor Jonathan Katz
Professor John Dickerson
Professor Dana Dachman-Soled
Professor Lawrence C. Washington, Dean's Representative

© Copyright by
Shravan Srinivasan
2023

To my loved ones, thank you for your encouragement

and support during this ride.

Acknowledgments

Graduate school has been the ride of a lifetime, and I am thankful for all the support I got along the way.

Thanks to my adviser, **Charalampos (Babis) Papamanthou**, without him this dissertation would not have been possible. I am indebted to his indispensable guidance, steadfast support, and patience throughout my Ph.D. journey. Babis has always been available to discuss research and provide valuable feedback. I am immensely grateful for the opportunity to work with him.

Thanks to other members of my dissertation committee: **Jonathan Katz**, **John Dickerson**, **Dana Dachman-Soled**, and **Lawrence C. Washington**, for the feedback, questions, and discussions that helped me along the way. Thanks to Jon Katz for additionally helping me navigate unexpected administrative challenges.

Thanks to **Alin Tomescu** for his infectious enthusiasm, meticulous attention to detail, and hilarious stand-up material that made our meetings truly memorable. I hope to emulate his ability to write clearly and make terrific presentations in my future endeavors. Thanks to **Kartik Nayak** for his positive attitude and zen-like presence in our meetings. I am grateful for the wealth of knowledge he has shared with me, ranging from invaluable pointers on navigating graduate school to strategizing and winning UMD CS game nights.

Thanks to my other wonderful coauthors: **Yupeng Zhang**, **Julian Loss**, **Sri AravindaKrishnan Thyagarajan**, **Giulio Malavolta**, **Foteini Baldimtsi**, **Ioanna Karan-**

taidou, and **Alexander Chepurnoy**. Their collaboration has played a pivotal role in completing this dissertation. Thanks to my internship mentors and colleagues: **Elaine Shi**, **Chris Peikert**, and **Leo Fan**, for the exciting industry experience and countless memorable lunches.

Thanks to **Arunchandar Vasan** and **Venkatesh Sarangan** for their mentorship and support. They gave me the toolset and the confidence to pursue research. Without them, I wouldn't have considered doing a Ph.D.

Thanks to **Ahmed Kosba**, **Ioannis Demertzis**, **Giorgos Tsimos**, and all other lab mates for creating an amazing lab environment.

Thanks to **Tom Hurst**, **Dana Purcell**, **Emily Hartz**, **Sharron McElroy**, and **Vivian Lu** for graciously helping me all these years. Their superpowers in tackling bureaucratic and administrative challenges saved me countless times during my Ph.D. Thanks to **Sam Handwerger** for his invaluable expertise in tax codes. His timely guidance was instrumental in navigating some peculiar situations that I encountered.

Thanks to all my friends for the fantastic company, great times, delicious food, and, well, for keeping me sane: **Hari**, **Nishant**, **Devesh**, **Ananth**, **Mack**, **Yogesh**, **Ajay**, **Heba**, **Flores**, and many others.

Thanks to **Anirudh** and **Nitin** for the home away from home. Thanks to **Preethi** for her continuous support and invaluable phone calls that kept my spirits high during challenging times. Thanks to my **parents** for their constant encouragement, countless sacrifices, and the opportunities they provided. Thanks to **Nikita** for her love, unconditional support, and always having my back through the highs and lows of my life. I am forever grateful for her presence in my life.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents v

List of Tables viii

List of Figures x

Chapter 1: Introduction 1
- 1.1 Overview of contributions . . . 3
  - 1.1.1 Authenticated data structures . . . 3
  - 1.1.2 Consensus protocols . . . 4
- 1.2 Publications . . . 5

Chapter 2: Preliminaries 6
- 2.1 Cryptographic assumptions . . . 6
- 2.2 Time-Lock Puzzles . . . 8
- 2.3 Time-Lock Puzzles with Batch Solving . . . 10

I Data Structures for Scalability in Distributed Consensus 12

Chapter 3: Aggregatable and Maintainable Vector Commitment 13
- 3.1 Overview of Techniques . . . 16
- 3.2 Related work . . . 19
- 3.3 Preliminaries . . . 21
  - 3.3.1 Multilinear extension (MLE) of a vector . . . 22
  - 3.3.2 PST commitments to MLEs . . . 24
  - 3.3.3 Vector Commitments (VCs) . . . 26
  - 3.3.4 Inner Product Arguments (IPA) . . . 28
- 3.4 Hyperproofs . . . 29
  - 3.4.1 Multilinear trees (MLTs) . . . 30
  - 3.4.2 Updates and homomorphism . . . 32
  - 3.4.3 Aggregating proofs . . . 35
  - 3.4.4 Unstealable proofs . . . 39

3.5 Hyperproofs for Cryptocurrencies . . . 44
3.6 Evaluation . . . 47
3.6.1 Microbenchmarks . . . 48
3.6.2 Comparison with SNARKs . . . 50
3.6.3 Macrobenchmarks . . . 54
3.7 Discussion . . . 58
3.8 Assumptions, definitions, and primitives . . . 60
3.9 VCs and Hyperproofs . . . 65
3.9.1 Unstealable VCs . . . 68

Chapter 4: Batching, Aggregation, and Zero-Knowledge Proofs in Bilinear Accumulators 71
4.1 Related work . . . 78
4.2 Preliminaries . . . 81
4.2.1 Cryptographic accumulators . . . 84
4.2.2 Correctness and soundness . . . 85
4.2.3 Accumulator based on bilinear-maps . . . 87
4.3 Batching BP accumulator proofs . . . 89
4.3.1 Membership . . . 89
4.3.2 Non-membership . . . 90
4.4 Aggregation . . . 91
4.4.1 Membership . . . 91
4.4.2 Non-membership . . . 93
4.5 Non-Membership proof updates . . . 96
4.6 Proof of Exponentiation . . . 99
4.6.1 Soundness of PoE . . . 102
4.7 ZK batch proofs . . . 103
4.7.1 Proving membership ($\mathcal{R}_{\mathsf{mem}}$) . . . 104
4.7.2 Proving non-membership ($\mathcal{R}_{\mathsf{nonmem}}$) . . . 106
4.7.3 Proving degree bound ($\mathcal{R}_{\mathsf{degcheck}}$) . . . 107
4.8 Evaluation . . . 110
4.8.1 Aggregation . . . 111
4.8.2 Zero-knowledge batch proofs . . . 117
4.8.3 Comparison with HARiSA [43] . . . 119
4.9 Example: Non-membership aggregation . . . 121

II Protocols to Improve Security in Distributed Consensus 122

Chapter 5: Permissionless Protocol in the Mobile Sluggish Model 123
5.1 Technical overview . . . 125
5.2 Attack on Nakamoto consensus in the mobile sluggish model . . . 128
5.3 Model . . . 130
5.4 Protocol . . . 132
5.5 Analysis . . . 133

5.6 Related work 135

Chapter 6: Byzantine Broadcast under Strongly Adaptive Adversaries 138
6.1 Technical overview 139
6.2 Model and definitions 140
6.3 Protocol 143
6.4 Analysis 145
6.5 Related Work 150

Chapter 7: Conclusion and Future Directions 152
7.1 Conclusion 152
7.2 Future directions 153

Bibliography 155

List of Tables

3.1 Comparison with other VCs, which are *not* simultaneously *aggregatable* and *maintainable* (see "Agg time" and "UpdAllProofs time" columns). n is the size of the vector, π_i is a proof for position i and π_I is an aggregated proof for k positions. Proof sizes and time complexities are in terms of group elements and group exponentiations / field operations, respectively. (In RSA-based VCs [26, 42, 46, 90], we count $O(\ell)$ group operations as an exponentiation, where ℓ is the bit-width of VC elements.) Items in red indicate worse performance than Hyperproofs. All schemes* support UpdDig and UpdProof (see Definition 3.3.1). 18
3.2 Single-threaded microbenchmarks for Hyperproofs. Running times with an asterisk symbol (*) are too long and have been interpolated. We measure aggregation of 1024 proofs. OpenAll and Com are only measured once. UpdDig and UpdAllProofs times are averages after a batch of 1024 changes to the vector. All algorithms are parallelizable. 47
3.3 The size of the public parameters from Fig. 3.1 for various values of $\ell = \log_2 n$. Recall that the *verification key* consists of all selector monomial commitments $g_2^{s_k}, \forall k \in [\ell]$, while the *proving key* consists of all selector multinomial commitents $g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0, \ell], j \in [0, 2^k)$ (see Fig. 3.1). 48
3.4 Single-threaded, stateless cryptocurrency macrobenchmarks that measure the time to prepare a block for proposal (**P**), to validate a proposed block (**V**) and to update all proofs (**M**) after a new block is seen. A block propagates through a P2P network of diameter $h = 20$. Trees have height $\ell = 30$ and blocks have 1024 transactions. A Poseidon-128 hash takes 113 μs using the `go-iden3-crypto` library [71]. A Pedersen hash takes 37 μs using the `sapling-crypto` library [118]. 57

4.1 Set X denotes the entire accumulated set and $I \subseteq X$. Let $O(Y) \cdot \mathbb{G}$ denotes one large exponentiation to the product of Y elements or Y exponentiations. All exponentiations in BP can be sped up by a logarithmic factor using multi-exponentiations. 96
4.2 Accumulator batching operation costs for different batch sizes. In the first column, (s) denotes seconds and (min) minutes. The costs for RSA operations include the computations required to map to the prime domain. N/A stands for very large costs which are not interesting to compute. 111
4.3 Sizes of accumulator digest and proofs in bytes. Asterisk(*) denotes a batch size of 2^{17}. 115
4.4 Single-threaded microbenchmarks for our ZK constructions. 117

4.5 Verification overhead and proof size. 121

List of Figures

3.1 PST (and Hyperproofs) public parameters. The uth path in this tree is actually the update key upk_u from Eq. (3.16). 26
3.2 $O(n)$-time algorithm for computing a single PST evaluation proof π_i for $f(\mathbf{i})$ w.r.t. an MLE f of size $n = 2^\ell$. 27
3.3 A multilinear tree (MLT) of size 8. Recall from Section 3.3 that $f_{b_\ell b_{\ell-1} \dots b_k}$ denotes the MLE of $\mathbf{a}_{b_\ell b_{\ell-1} \dots b_k}$. Each node stores a PST commitment to the depicted MLE: e.g., root stores $\mathsf{pst}(f_1 - f_0)$, not $f_1 - f_0$. The proof for a_i consists of all commitments along a_i's path to the root (e.g., for a_4, the boxed nodes). Sibling leaves $[a_{2j}, a_{2j+1}]$ have the same proof. If, say, a_4 changes, all pink-colored MLEs change, and all boxed commitments must be updated. 31
3.4 Computes an MLT in $O(n \log n)$ time consisting of PST evaluation proofs for *all* $f(\mathbf{i})$ w.r.t. an MLE f of $\mathbf{a}$ of size $n = 2^\ell$. In contrast, n naive calls to $\mathsf{PST.Prove}$ would take $O(n^2)$. Recall that f_0 and f_1 are MLEs for the left and right halves of $\mathbf{a}$. Returns the tree stored in preorder in an array. . . . 32
3.5 Our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ used to aggregate Hyperproofs. H is a random oracle and $(\mathcal{G}_{\mathsf{IPA}}, \mathcal{P}_{\mathsf{IPA}}, \mathcal{V}_{\mathsf{IPA}})$ is the Bünz et al. IPA from Section 3.3.4. . . . 36
3.6 Algorithms for Hyperproofs, implicitly parameterized by the max number of proofs b that can be aggregated into a single proof. 38
3.7 SNARK-based Merkle proof aggregation versus Hyperproof aggregation. The x-axis is $\log_2$(# of proofs being aggregated). Dotted lines are extrapolated, due to the SNARK prover running out of memory. We use the 128-bit secure variant of Poseidon. 51
3.8 The *interactive* IPA by Bünz et al. for $m = 2^k$ (wlog). The prover convinces the verifier that he knows $(\mathbf{A}, \mathbf{B}) \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ such that $\mathbf{A}, \mathbf{B}, Z$ are committed in $\mathbf{C}$ (under commitment key $\mathbf{ck}$) and that $Z = \langle \mathbf{A}, \mathbf{B} \rangle$. See Section 3.3 for IPA-specific notation such as $1_{\mathbb{G}_b}, \mathbf{A}_L, \mathbf{A} \circ \mathbf{B}, \mathsf{CM}, \mathbf{A}_R || 1_{\mathbb{G}}$ or $\mathbf{A}_L^x$. 64

4.1 PoE protocol. We use Fiat-Shamir transformation to make this protocol into non-interactive (Fig. 4.2). For the ease of exposition we present the construction in the symmetric pairing setting. However, we remark that our implementation uses asymmetric paring. 100
4.2 Non-interactive PoE protocol. 101
4.3 We extrapolate the proving costs using the numbers reported in HARiSA [43]. Note that the results in the RSA setting does include the Hash-to-prime costs. 118

5.1 Double spend attack: This plot depicts average block arrival times. Assuming 52 honest nodes (51 prompt + 1 sluggish) and 48 adversarial nodes, the average inter-arrival time of honest blocks and adversarial blocks in Bitcoin is 19.2 and 20.8 minutes, respectively. Observe that over 19.2×2 minutes, even though the honest nodes have mined two blocks, due to sluggishness, the honest chain has grown only by one block. 129

6.1 Timing of execution and messages in the simulation by $\mathcal{B}$ 150

Chapter 1: Introduction

Distributed consensus is the problem of reaching an agreement among mutually distrusting nodes in the presence of faults. In 2008, Bitcoin whitepaper proposed Nakamoto consensus, the first *state machine replication* algorithm (SMR) to achieve consensus on an internet-scale and in a permissionless setting [100]. At the core of the protocol is the ever-growing, append-only ledger called the blockchain. Every node in the system maintains the entire blockchain, executes all the transactions recorded in the blockchain, and tries to extend the blockchain. Since its inception, blockchains have found various applications ranging from cryptocurrencies to digitally bound contracts. Despite the rising popularity, blockchains suffer scalability and security challenges.

Scalability. Numerous on-chain and off-chain efforts aim to improve the scalability of blockchains [39, 59, 70, 115, 117]. *Sharding* is one such proposal that seeks to securely scale blockchains without compromising decentralization [81, 150, 159]. In sharding, nodes are partitioned into small committees, and each committee maintains only a part of the blockchain rather than all nodes maintaining the entire blockchain. Moreover, nodes have to be frequently shuffled between shards to prevent adversaries from corrupting a majority of nodes within a shard. In the quest to improve scalability and security, Ethereum in its upcoming upgrade is set to deploy sharding in the production environment [38, 47]. Scaling

through sharding is orthogonal to other on-chain and off-chain scaling efforts. Therefore, these scaling techniques can be combined to improve scalability.

Existing blockchains rely on the *validation state* to verify new blocks and transactions. Unfortunately, this validation state can become very large (e.g., Ethereum's validation state has reached hundreds of gigabytes), which is a challenge for sharding [26, 35, 50]. Large validation state: (1) impedes scalability as verifying blocks and transactions can be slow when the state is stored on the disk, (2) slows down sharded consensus protocols as while switching shards, a validator has to download the new validation state for the new shard, and (3) downloading and storing the validation state can be a barrier to entry for consumer machines. Stateless blockchains propose to remove the burden of storing the validation state and allow a node to participate in the consensus with a constant size memory. Thus, in order to take advantage of sharding, blockchains have to become *stateless*.

Security. A blockchain protocol (or classically referred as SMR protocol) must satisfy two security properties: *consistency* and *liveness*. Consistency ensures that all honest nodes have the same view of the blockchain, and liveness ensures that honest nodes will extend the blockchain at a steady rate. Blockchain protocols have been studied under various settings and assumptions. However, the ability of an adversary to delay or delete messages is modeled as the *network* (reliability of the communication) and *adaptivity* of the adversary (ability corrupt honest nodes on the fly).

Garay et al. showed that Nakamoto consensus has consistency and liveness under permissionless setting as long as the network is *synchronous* (message sent by an honest node reaches other nodes in a known amount of time) even if the adversary can *adaptively*

corrupt honest nodes on the fly [66, 112, 120]. *Asynchronous* setting (messages can take an arbitrary amount of time to reach) and *partially synchronous* setting (messages will be synchronous after the network stabilizes) relaxes the synchronous setting and closely models the internet. However, Pass and Shi showed that it is impossible to achieve permissionless consensus in an asynchronous or partially synchronous network regardless of the adaptivity of the adversary [114, Theorem 4]. Unfortunately, expecting that the network is synchronous for a protocol deployed on the internet is exceedingly optimistic.

1.1 Overview of contributions

The contributions of this dissertation can be categorized into: (1) authenticated data structures to improve scalability in blockchains, and (2) secure consensus protocols to handle powerful network-level adversaries.

1.1.1 Authenticated data structures

Vector commitment (Chapter 3): We present Hyperproofs, the first vector commitment (VC) scheme that is efficiently *maintainable* and *aggregatable* [128]. Hyperproofs is also *unstealability*, a novel property that incentivizes proof computation. Similar to Merkle proofs, our proofs form a tree that can be efficiently maintained: updating all n proofs in the tree after a single leaf change only requires $O(\log n)$ time. Importantly, unlike Merkle proofs, Hyperproofs are *efficiently* aggregatable, anywhere from $10\times$ to $41\times$ faster than SNARK-based aggregation of Merkle proofs. At the same time, an individual Hyperproof consists of only $\log n$ algebraic hashes (e.g., 32-byte

elliptic curve points) and an aggregation of b such proofs is only $O(\log{(b \log n)})$-sized. Thus, making Hyperproofs useful in stateless blockchains.

Accumulator (Chapter 4): We propose algorithms that allow a prover to aggregate multiple individual non-membership accumulator proofs, in the Bilinear Pairings (BP) setting, into a single batch proof of constant size [129]. Additionally, we propose a novel *Proof-of-Exponentiation* (PoE) protocol in the BP setting to delegate the cost of exponentiation to an untrusted prover. Finally, we implement and evaluate a zero-knowledge batch proof with constant proof size and constant verification in the BP setting. Our scheme is around 16× to 42× faster than state-of-the-art SNARK-based zero-knowledge batch proofs in the RSA setting. Our work aims to reduce the communication size, improve verifier efficiency, and support privacy in the batch setting, thus making our techniques potentially useful in blockchain interoperability, consensus, stateless blockchain, and other applications.

1.1.2 Consensus protocols

We study the effects of delaying or deleting messages from the network on a consensus protocol's resilience and round complexity in the following settings:

Permissionless consensus (Chapter 5): We first show an attack to illustrate that Nakamoto consensus is not secure *even* in the mobile sluggish model, a weak model of synchrony where the adversary can adaptively create minority partitions in the network to delay messages. We then present a proof-of-work based permissionless protocol (Section 5.4) which does not assume that the network is synchronous or all

honest messages arrive on time [132]. To the best of our knowledge, this is the *first* work to show that it is possible to achieve consensus in the permissionless setting even after *relaxing the standard synchrony assumption!*

Byzantine broadcast (Chapter 6): We present the first *expected constant round* Byzantine broadcast under strongly adaptive and corrupt majority setting [132]. To realize our result, we develop a generic compiler to *convert any* broadcast protocol secure against a *weakly* adaptive adversary into a broadcast protocol secure against a *strongly* adaptive adversary in a *round preserving* way.

1.2 Publications

The results of this dissertation are also published as conference papers [128, 129, 132]:

"Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments,"
Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang, in USENIX Security 2022.

"Batching, Aggregation, and Zero-Knowledge Proofs in Bilinear Accumulators,"
Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou, in ACM CCS 2022.

"Transparent Batchable Time-lock Puzzles and Applications to Byzantine Consensus,"
Shravan Srinivasan, Julian Loss, Giulio Malavolta, Kartik Nayak, Charalampos Papamanthou, and Sri AravindaKrishnan Thyagarajan, in PKC 2023.

Chapter 2: Preliminaries

2.1 Cryptographic assumptions

In this section, we present the necessary cryptographic assumptions. Let BilGen be a randomized polynomial time algorithm that takes a security parameter λ as input and returns an asymmetric pairing instance $\mathsf{bp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ as the output. Let $\mathsf{negl}(\cdot)$ denote a negligible function and $\leftarrow\!\!\$$ represents sampling uniformly at random.

Assumption 2.1.1 (t-Strong Bilinear Diffie-Hellman Assumption (t-SBDH) [73]). *For any PPT adversary $\mathcal{A}$, the following probability:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda) \\ s \leftarrow\!\!\$\ \mathbb{Z}_p^* \\ \sigma = (\mathsf{bp}, \{g_1^{s^i} \mid 0 \leq i \leq t\}, g_2^s) \\ (\ell, e(g_1, g_2)^{\frac{1}{s+\ell}}) \leftarrow \mathcal{A}(1^\lambda, \sigma) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

The t-SBDH assumption is implied by the t-strong Diffie-Hellman assumption (t-sDH) [73]. t-sDH is hard in the Generic Group Model (GGM) as shown in [25].

In the RSA setting, Wesolowski et al. [151] introduce the *Adaptive root assumption* to prove soundness of their Verifiable Delay Function (VDF) construction. Informally, it

states that it is hard to find a *random root* of *any* group element. Similarly, we introduce the adaptive variant of the t-SBDH as follows:

Assumption 2.1.2 (t-strong Adaptive Bilinear Diffie-Hellman Assumption (t-SABDH))**.** *For any* PPT *adversary* $(\mathcal{A}_1, \mathcal{A}_2)$*, the following probability:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda) \\ s \leftarrow\!\!\$\ \mathbb{Z}_p^* \\ \sigma = (\mathsf{bp}, \{g_1^{s^i} \mid 0 \leq i \leq t\}, g_2^s) \\ (w \in \mathbb{G}_1 \setminus \{1_{\mathbb{G}_1}\}, \mathsf{state}) \leftarrow \mathcal{A}_1(1^\lambda, \sigma) \\ \ell \leftarrow\!\!\$\ \mathbb{Z}_p^* \setminus \{-s\} \\ e(w, g_2)^{\frac{1}{s+\ell}} \leftarrow \mathcal{A}_2(\sigma, \ell, \mathsf{state}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Informally, Assumption 2.1.2 argues that for *any* group element chosen by the adversary (excluding the identity element) and a *randomly* chosen ℓ, it is hard to compute the $(s+\ell)$-th root of the element $e(w, g_2)$. In contrast, Assumption 2.1.1 argues that for *any* ℓ, it is hard to compute the $(s+\ell)$-th root of the element $e(g_1, g_2)$ [73].

Assumption 2.1.3 (t-power knowledge-of-exponent (t-PKE) assumption)**.** *For any* PPT *adversary* $\mathcal{A}$*, there exists a* PPT *extractor* $\mathcal{X}_\mathcal{A}$ *[75]:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda) \\ a, s \leftarrow\!\!\$\, \mathbb{Z}_p^* \\ \mathsf{pp} = (\mathsf{bp}, \{g_j^{s^i}, g_j^{as^i} \mid 0 \leq i \leq t, j \in \{0,1\}\}) \\ (c, \hat{c}; a_0, \ldots, a_t) \leftarrow (\mathcal{A}||\mathcal{X}_\mathcal{A})(\mathsf{pp}) : \\ \hat{c} = c^a \wedge c \neq \prod_{i=0}^{t} g_j^{a_i s^i} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Where $(c, \hat{c}; a_0, \ldots, a_t) \leftarrow (\mathcal{A}||\mathcal{X}_\mathcal{A})(\mathsf{pp})$ *denotes that on input* pp*,* $\mathcal{A}$ *outputs* $c, \hat{c}$ *and on the same input* $\mathcal{X}_\mathcal{A}$ *outputs* $a_0, \ldots, a_t$*.*

2.2 Time-Lock Puzzles

A *Time-Lock Puzzle* (TLP) is a cryptographic primitive that allows a sender to lock a message as a computational puzzle in a manner where the receiver will be able to solve the puzzle after a stipulated time $\mathbf{T}$. In terms of efficiency, a sender should be able to generate a puzzle substantially faster than the time required to solve it, and in terms of security, an adversary should not be able to solve the puzzle faster than the stipulated time, even with parallel computation. Rivest, Shamir, and Wagner (RSW) [122] proposed the first TLP construction based on the sequentiality of repeated modular squaring in the RSA group. Many other TLP constructions [23, 95, 137] have followed suit in different settings but require the same flavor of sequential operations during solving.

We give a formal definition of TLPs [122]. The syntax follows the standard notation for TLPs except that we consider an additional setup phase that depends on the hardness parameter but not on the secret.

Definition 2.2.1 (Time-Lock Puzzles). *Let $\mathcal{S}$ be a finite domain. A time-lock puzzle (TLP) with solution space $\mathcal{S}$ is tuple of four algorithms* $(\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PSol})$ *defined as follows.*

- $\mathsf{pp} \leftarrow \mathsf{PSetup}(1^n, \mathbf{T})$ *a probabilistic algorithm that takes as input a security parameter 1^n and a time hardness parameter $\mathbf{T}$, and outputs public parameters* pp.

- $Z \leftarrow \mathsf{PGen}(\mathsf{pp}, s)$ *a probabilistic algorithm that takes as input public parameters* pp*, and a solution $s \in \mathcal{S}$, and outputs a puzzle Z.*

- $s \leftarrow \mathsf{PSol}(\mathsf{pp}, Z)$ *a deterministic algorithm that takes as input public parameters* pp *and a puzzle Z and outputs a solution s.*

Definition 2.2.2 (Correctness). *A TLP scheme* $(\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PSol})$ *is correct if for all $\lambda \in \mathbb{N}$, all polynomials $\mathbf{T}$ in λ, all secrets $s \in \mathcal{S}$, and all* pp *in the support of* $\mathsf{PSetup}(1^n, \mathbf{T})$*, it holds that:* $\Pr[\mathsf{PSol}(\mathsf{pp}, \mathsf{PGen}(\mathsf{pp}, s)) = s] = 1$.

Security requires that the solution of the puzzles is hidden for all adversaries that run in (parallel) time less than $\mathbf{T}$.

Definition 2.2.3 (Security). *A TLP scheme* $(\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PSol})$ *is secure with gap $\varepsilon < 1$ if there exists a polynomial $\tilde{\mathbf{T}}(\cdot)$ such that for all polynomials $\mathbf{T}(\cdot) \geq \tilde{\mathbf{T}}(\cdot)$ and every polynomial-size adversary $(\mathcal{A}_1, \mathcal{A}_2) = \{(\mathcal{A}_1, \mathcal{A}_2)_n\}_{n \in \mathbb{N}}$ where the depth of $\mathcal{A}_2$ is bounded from above by $\mathbf{T}^{\varepsilon}(n)$, there exists a negligible function $\mu(\cdot)$, such that for all $n \in \mathbb{N}$ it holds that*

$$\Pr\left[\begin{array}{c} b \leftarrow \mathcal{A}_2(\mathsf{pp}, Z, \mathsf{st}) \\ \wedge\, (s_0, s_1) \in \mathcal{S}^2 \end{array} : \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{PSetup}(1^n, \mathbf{T}(n)) \\ (\mathsf{st}, s_0, s_1) \leftarrow \mathcal{A}_1(1^n, \mathsf{pp}) \\ b \leftarrow\!\!\$\; \{0,1\}, Z \leftarrow \mathsf{PGen}(\mathsf{pp}, s_b) \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

2.3 Time-Lock Puzzles with Batch Solving

In many TLP applications involving multiple users, it is often the case that a user is required to solve the puzzles of all other users, and record all of the solutions. Say an auction house has to open all the time-locked bids and declare them publicly before announcing the winner. Batching and solving the puzzles is essential for scalability in such TLP applications that have large number of participating users. Intuitively, batch solving of TLPs allows a receiver to solve multiple puzzles simultaneously (at the price of solving one puzzle) without needing to solve each puzzle separately. Specifically, the total running time of the batch-solve operation is bounded by some $p(\lambda, \mathbf{T}) + \tilde{p}(\lambda, n)$ for some fixed polynomials p and $\tilde{p}$, where λ is the security parameter, n is the number of puzzles to be batched, and $\mathbf{T}$ is the timing hardness of a single puzzle.

We define the notion of TLPs with batched solving. We borrow the standard interfaces of a TLP from Section 2.2 and append it with an interface to allow for batched solving of n puzzles.

Definition 2.3.1 (Batch Solving)**.** *A TLP scheme* $(\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PSol})$ *supports* batch solving *with the aid of an additional interface defined below*

- $(s_1, \ldots, s_n) \leftarrow \mathsf{BatchPSol}(\mathsf{pp}, Z_1, \ldots, Z_n)$ *a deterministic algorithm that takes as input public parameters* pp *and puzzles* $Z_1, \ldots, Z_n$*, and outputs solutions* $s_1, \ldots, s_n$*.*

Definition 2.3.2 (Batch Solving Correctness)**.** *An TLP scheme* $(\mathsf{PSetup}, \mathsf{PGen}, \mathsf{PSol})$ *with batch solving interface* $\mathsf{BatchPSol}$ *is correct if for all* $n \in \mathbb{N}$*, all polynomials* $\mathbf{T}$ *in* n*, all*

polynomials n *in* n*, all solutions* $(s_1, \ldots, s_n) \in \mathcal{S}^n$*, all* pp *in the support of* $\mathsf{PSetup}(1^n, \mathbf{T})$*, and all* Z_i *in the support of* $\mathsf{PGen}(\mathsf{pp}, s_i)$*, the following conditions are satisfied:*

- *There exists a negligible function* $\mu(\cdot)$ *such that*

$$\Pr\left[\mathsf{BatchPSol}(\mathsf{pp}, Z_1, \ldots, Z_n) \neq (s_1, \ldots, s_n)\right] \leq \mu(n).$$

- *There exist fixed polynomials* $p(\cdot), \tilde{p}(\cdot)$ *such that the size complexity of the circuit evaluating* $\mathsf{BatchPSol}(\mathsf{pp}, Z_1, \ldots, Z_n)$ *is bounded by* $p(\lambda, \mathbf{T}) + \tilde{p}(\lambda, n)$*.*

Notice that the above definition rules out trivial solutions, where you solve the n puzzles individually and output the solutions. This is because, in this solution the size scales with $n \cdot \mathbf{T}$, while the definition above only permits the scale to be $n + \mathbf{T}$. One can view $\tilde{p}(\lambda, n)$ as capturing the time taken to read and process the n puzzles, and returning the n solutions. The factor $p(\lambda, \mathbf{T})$ captures the solving of a single puzzle and itself is independent of n.

Part I

Data Structures for Scalability in Distributed Consensus

Chapter 3: Aggregatable and Maintainable Vector Commitment

Vector commitment (VC) schemes [46,93] such as Merkle trees [96] are fundamental building blocks in many protocols. In a VC scheme, a *prover* computes a succinct *digest* d of a vector $\mathbf{a} = [a_1, \ldots, a_n]$ and proofs $\pi_1, \ldots, \pi_n$ for each position. A *verifier* who has the digest d can later verify a proof π_i that a_i is the correct value at position i. Some VCs, such as Merkle trees, are *maintainable*: when the vector changes *all* proofs can be *efficiently* updated in sublinear time, rather than recomputed from scratch in linear time. Other VCs, such as Pointproofs [72], are *aggregatable*: the prover can take several proofs π_i for $i \in I$ and *efficiently* aggregate them into a single, succinct proof π_I.

Vector commitment (VC) schemes [46,93] such as Merkle trees [96] are fundamental building blocks in many protocols. In a VC scheme, a *prover* computes a succinct *digest* d of a vector $\mathbf{a} = [a_1, \ldots, a_n]$ and proofs $\pi_1, \ldots, \pi_n$ for each position. A *verifier* who has the digest d can later verify a proof π_i that a_i is the correct value at position i. Some VCs, such as Merkle trees, are *maintainable*: when the vector changes *all* proofs can be *efficiently* updated in sublinear time, rather than recomputed from scratch in linear time. Other VCs, such as Pointproofs [72], are *aggregatable*: the prover can take several proofs π_i for $i \in I$ and *efficiently* aggregate them into a single, succinct proof π_I.

Unfortunately, no current VC scheme is both maintainable and aggregatable; at least not efficiently. Yet emerging applications such as *stateless cryptocurrencies* [26, 57, 72, 98, 121, 138, 141] rely on dedicated nodes to efficiently maintain all proofs and also on miners to efficiently aggregate proofs. While generic argument systems (e.g., SNARKs [76, 111]) can be used to add aggregation to maintainable VCs such as Merkle trees, this is too slow in practice (see Section 3.6.2). Thus we ask: *Can we build an efficient VC that is both maintainable and aggregatable?* In this work, we answer this positively and present *Hyperproofs*. Similar to Merkle trees, Hyperproofs are $\log n$-sized and determine a tree. This makes updating all proofs very efficient in logarithmic time. However, Hyperproofs are built from *polynomial commitments* [82, 108] rather than hash functions such as SHA-256. This enables a natural aggregation algorithm that is $10\times$ to $41\times$ faster than "SNARKing" multiple Merkle proofs.

In addition to aggregation and maintainability, Hyperproofs have another very useful property: *homomorphism*. Specifically, trees (and digests) for two vectors can be combined into a single tree (and digest) for their sum. This has several applications. First, homomorphism allows us to obtain *unstealability*, a property which incentivizes proof computation in applications such as stateless cryptocurrencies [151]. In a nutshell, unstealability allows a prover to *watermark the proofs she computes with her identity, in an irreversible manner*. This way, honest provers can be rewarded for the proofs they compute while malicious provers cannot *steal* other provers' proofs. Second, homomorphism makes updating digests (and Hyperproofs) more convenient than updating Merkle roots (and proofs), which requires having the proof(s) for the changed position(s) in the vector. Third, homomorphism allows authenticating data in a streaming setting [109].

Challenges. In designing Hyperproofs, we surmount three key challenges. First, computing n proofs in Papamanthou-Shi-Tamassia (PST) polynomial commitments [108] takes $O(n^2)$ time and is too slow. Second, aggregation of PST proofs is difficult without generic SNARKs [76, 111], which would be too expensive. Third, unstealable proofs must remain maintainable and aggregatable. This precludes solutions based on computing SNARKs over proofs which, in addition to being slow (see Section 3.6.2), would sacrifice updatability (see *"Strawmen"* in Section 3.4.4). Furthermore, unstealable proofs must continue to verify with respect to one global digest. This precludes solutions that embed the identity of the prover inside the vector, which results in as many digests as there are provers (and would only be practical in a small-scale, permissioned setting).

Evaluation. In Section 3.6.1, we show Hyperproofs are small (1.44 KiB), they verify quickly (17.4 milliseconds) and are fast to maintain (2.6 milliseconds per update). In Section 3.6.2, we show Hyperproof aggregation is much faster than Merkle proof aggregation: $10\times$ faster when using Poseidon hashes [74], which likely need more cryptanalysis, and $41\times$ faster when using provably-secure Pedersen hashes. However, our faster aggregation comes at a cost of slower verification for aggregated proofs and a larger 52 KiB aggregate proof size. Nonetheless, when considering the end-to-end aggregation and verification time in stateless cryptocurrencies, Hyperproofs remain $10\times$ to $41\times$ faster and outperform Merkle trees (see Section 3.6.3).

Limitations. To commit to a vector of size n, Hyperproofs requires public parameters consisting of $2n-1$ group elements, which must be generated via a *trusted setup*, typically decentralized via multi-party computation protocols [30]. In future work, we hope to have a

transparent setup by using assumptions in hidden-order groups. We also do not explore the subtleties of fully-integrating unstealable proofs into a statelessly-validated cryptocurrency. Lastly, our macrobenchmarks only measure the computational overhead of VCs that arises on the critical path to a consensus decision. While our results show Hyperproofs lead to $10\times$ faster decisions, we do not claim this is sufficient to make the stateless setting practical.

3.1 Overview of Techniques

Vectors as multilinear extensions (MLEs). We build upon previous work [162, 163] that represents a vector of size $n = 2^\ell$ as a *multilinear extension (MLE)* polynomial. For example, the MLE of $\mathbf{a} = [5, 2, 8, 3]$ is $f(x_2, x_1) = 5(1 - x_2)(1 - x_1) + 2(1 - x_2)x_1 + 8x_2(1 - x_1) + 3x_2x_1$. Note that f correctly "selects" the right a_i given the binary expansion of i as input: $f(0, 0) = 5$, $f(0, 1) = 2$, $f(1, 0) = 8$ and $f(1, 1) = 3$.

PST commitments to MLEs. To commit to a vector, we compute a Papamanthou-Shi-Tamassia (PST) commitment [108] to its MLE (see Section 3.3.2) . For example, the PST commitment to f above is $\mathsf{C} = g_1^{f(s_1,s_2)} \in \mathbb{G}_1$, where $(s_1, s_2) \in \mathbb{Z}_p^2$ are secret points encoded in the *public parameters* of the scheme and g_1 is the generator of $\mathbb{G}_1$. For vectors of size 4, these public parameters consist of $g_1^{s_1}, g_1^{1-s_1}, g_1^{(1-s_2)(1-s_1)}$, $g_1^{(1-s_2)s_1}, g_1^{s_2(1-s_1)}, g_1^{s_2 s_1}$. Importantly, we show that the selectively-secure variant of PST commitments is actually adaptively-secure when restricted to only proving evaluations on the Boolean hypercube $\{0, 1\}^\ell$ (see Section 3.3.2). This reduces our proof size compared to previous work based on PST [162, 163].

Multilinear trees. To prove that a_i is the ith value in the vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]$, we compute a *PST evaluation proof* for $f(i_\ell, \ldots, i_1) = a_i$ w.r.t. the commitment C, where $(i_\ell, \ldots, i_1)$ is the binary representation of i. Unfortunately, this takes $O(n)$ time *per position.* Thus, computing all n proofs would take $O(n^2)$ time which is prohibitive. We reduce this to $O(n \log n)$ by computing a novel *multilinear tree (MLT)* of proofs using a divide-and-conquer approach. Importantly, our MLT is *maintainable*: updating all proofs after a change to the vector only requires $O(\log n)$ time.

Proof aggregation. A proof π_i for a_i consists of PST commitments $(w_{i,\ell}, \ldots, w_{i,1}) \in \mathbb{G}_1^\ell$ defined in Fig. 3.2, such that the following *pairing equation* holds:

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{j \in [\ell]} e(w_{i,j}, g_2^{s_j - i_j}), \tag{3.1}$$

where $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a *pairing* and $g_2^{s_j}$'s are additional $O(\ell)$-sized PST public parameters in $\mathbb{G}_2$. To aggregate b proofs, we prove *knowledge* of $w_{i,j}$'s that pass Eq. (3.1) for each i, resulting in a succinct $O(\log(b\ell))$ aggregated proof size. Our key ingredient is an *inner-product argument (IPA)* by Bünz et al. [34] for proving several pairing equations hold.

Homomorphism and unstealablity. As we mentioned, Hyperproofs are *homomorphic*: exponentiating a PST evaluation proof $(w_{i,\ell}, \ldots, w_{i,1})$ by a constant α yields a proof for position i but in a vector whose values are multiplied by α. We observe that if α is the secret key of a *proof-serving node (PSN)*, this makes the proof unstealable by other nodes who do not have α. Importantly, the proof can still be verified against the digest C, except the verifier must also give the node's corresponding public key g_2^α: $e(\mathsf{C}/g_1^{a_i}, g_2^\alpha) = \prod_{j \in [\ell]} e(w_{i,j}^\alpha, g_2^{s_j - i_j})$.

Table 3.1: Comparison with other VCs, which are *not* simultaneously *aggregatable* and *maintainable* (see "Agg time" and "UpdAllProofs time" columns). n is the size of the vector, π_i is a proof for position i and π_I is an aggregated proof for k positions. Proof sizes and time complexities are in terms of group elements and group exponentiations / field operations, respectively. (In RSA-based VCs [26, 42, 46, 90], we count $O(\ell)$ group operations as an exponentiation, where ℓ is the bit-width of VC elements.) Items in red indicate worse performance than Hyperproofs. All schemes* support UpdDig and UpdProof (see Definition 3.3.1).

Scheme	$\lvert\pi_i\rvert$	$\lvert\pi_I\rvert$	OpenAll time	Agg time	UpdAllProofs time	Trans-parent?	Homo-morphic?	Gen time	$\lvert$pp$\rvert$
AMT [139]	$\log n$	×	$n \log n$	×	$\log n$	×	✓	n^2	$n \log n$
aSVC [141]	1	1	$n \log n$	$k \log^2 k$	n	×	✓	$n \log n$	n
BBF [26]	1	1	$n \log^2 n$	$k \log n$	$n \log n$	×†	×	1	1
CF-CDH [46, 72, 90]	1	1	n^2	k	n	×	✓	n^2	n^2
CF-RSA [42, 46, 90]	1	1	$n \log n$	$k \log^2 k$	n	×†	✓	1	1
CFG+RSA [42]	1	1	$n \log^2 n$	$k \log k \log n$	n	×†	×	1	1
Lattice [109, 119]	$\log n$	×	n	×	$\log n$	✓	✓	1	$\log n$
Merkle	$\log n$	×	n	×	$\log n$	✓	×	1	1
Merkle SNARK	$\log n$	1	n	$k \log n \log (k \log n)$	$\log n$	×	×	1	1
Pointproofs [72]	1	1	$n \log n$	k	n	×	✓	n	n
Hyperproofs	$\log n$	$\log(k \log n)$	$n \log n$	$k \log n$	$\log n$	×	✓	n	n

†: BBF, CF-RSA and CFG+RSA avoid the trusted setup if instantiated using class groups of imaginary quadratic order, which are known to be slower than RSA groups.

*: Merkle trees, BBF and CFG+RSA require *dynamic* update hints, rather than *static* update keys, for digest and proof updates. Only the *weakly-binding* variant of CFG+RSA supports digest updates. CF-CDH and Pointproofs have $O(n)$-sized update keys, which can be too large for some applications.

As an optimization, proof-serving nodes can exponentiate the PST public parameters by α before computing proofs. This way, when computing a multilinear tree (MLT) with these parameters, all proofs are implicitly unstealable and the MLT remains maintainable.

3.2 Related work

Below, we relate our VC to previous work and summarize in Table 3.1.

Merkle trees. Our proofs consist of $\log n$ (algebraic) hashes and can be as small as Merkle proofs if using 256-bit elliptic curves [16]. However, Hyperproofs are orders of magnitude slower to compute and update, when compared to normal Merkle trees hashed with SHA-256. Nonetheless, when compared to aggregatable Merkle trees that use SNARK-friendly hash functions (e.g., Poseidon-128 [74]), Hyperproofs are only slightly slower to compute and update (see Section 3.6.3) but have faster aggregation, homomorphism and unstealability.

SNARK-based works. Ozdemir et al. [105] explore using SNARKs to prove knowledge of changes that update a vector with digest d into a new vector with digest d'. Lee et al. [91] also use SNARKs to prove correctness of state transitions in replicated state machines, without having to send the state changes. Neither work explores unstealability nor maintaining and aggregating proofs efficiently. Similar to our work, aggregating SNARK proofs [34] and some *proof-carrying data (PCD)* schemes [33] also rely on inner-product arguments.

Algebraic VCs. Zhang et al. [162,163] were the first to build VCs from PST commitments to MLEs. However, their $O(\log n)$-sized proofs are concretely larger and do not support up-

dates. Some VCs have $O(1)$-sized proofs [26,42,46,72,88,90,141], which inherently require $\Theta(n)$ time to update all proofs after a change. Aggregation and verification in these VCs is concretely, and sometimes asymptotically, faster (see Table 3.1). They also have smaller aggregated proofs. However, these VCs are not efficiently maintainable (see Section 3.6.1), which precludes using them in settings where provers are rewarded to maintain proofs (see Section 3.5).

Previous maintainable VCs [109,119,139,143] do not support aggregation; at least not without expensive generic argument systems (e.g., SNARKs). The lattice-based construction from [109,119] is also homomorphic and additionally transparent, with constant-sized public parameters. However, it is too slow for practice and non-aggregatable. The *authenticated multipoint evaluation tree (AMT)* construction from [139,143] can be viewed as the dual to our construction, but from univariate polynomials rather than multivariate. Unfortunately, it is non-aggregatable, its trusted setup requires $O(n^2)$ time and it has larger $O(n \log n)$-sized public parameters.

Recent work [8,26,144] enhances VCs into *key-value commitments (KVCs)*, where arbitrary keys (rather than vector positions) are mapped to values. Unfortunately, all of these constructions have constant-sized proofs and are thus not maintainable. Some VCs have transparent setup [26,42,90], support incremental aggregation [42], have a "specialiazable" CRS [42] and provide time/space trade-offs when computing proofs [26,42]. Hyperproofs do not have any of these features.

Unstealability. To the best of our knowledge, Katz et al. are the first to observe that (carefully) tying the identity of the prover to a proof she computes allows rewarding the

prover for her effort [83]. However, their work focuses on *watermarking zero-knowledge proofs of knowledge* of a secret witness. In contrast, in our work, our proofs need not be zero-knowledge and they need not prove knowledge of secret witnesses. Furthermore, unlike Katz et al.'s result, our notion of unstealability captures the difficulty of extracting useful information from watermarked proofs that might help an adversary steal proofs faster than computing them from scratch. Subsequently, Wesolowski explores such *watermarked proofs* in the context of verifiable delay functions (VDF) [151]. In contrast, we are the first to explore watermarking VC proofs and to give security definitions.

Although no previous VC scheme is unstealable, some can be made so using our pairing-based techniques from Section 3.4.4. Specifically, VCs from pairing-based polynomial commitments [72, 141, 143] appear compatible with our techniques. On the other hand, RSA-based VCs [26, 42, 46], which lack pairings, are less amenable to our techniques. While proofs-of-knowledge of exponent (PoKEs) [26] could be used to replace the reliance on pairings, this would come at the cost of losing maintainability of watermarked proofs. Lastly, our pairing-based techniques do not apply to Merkle trees as they are based on hash functions. Instead, we discuss watermarking Merkle proofs via SNARKs and their pitfalls in Section 3.4.4, under *"Strawmen"*.

3.3 Preliminaries

Notation. Let $[0, n) = \{0, 1, \ldots, n-1\}$. An ℓ-bit number i has *binary representation* $\mathbf{i} = (i_\ell, \ldots, i_1)$ if, and only if, $i = \sum_{k=0}^{\ell-1} i_{k+1} 2^k$. Note that i_ℓ is the MSB of i and i_1 is

the LSB. We often use $\mathbf{i}$ as i's binary representation and i_k as its kth bit, without explicit definition. Let $r \in_R S$ denote picking an element from S uniformly at random.

Pairings. $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$ denotes generating groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p, with g_i a generator of $\mathbb{G}_i$, and a *pairing* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that $\forall u \in \mathbb{G}_1, w \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p, e(u^a, w^b) = e(u, w)^{ab}$. A useful property of $e(\cdot, \cdot)$ is that $e(u, h)e(v, h) = e(uv, h), \forall u, v, h \in \mathbb{G}_1^2 \times \mathbb{G}_2$. In this work, we assume *Type III bilinear groups* (i.e., without efficiently-computable homomorphisms between $\mathbb{G}_1$ and $\mathbb{G}_2$ or viceversa), which are needed by the *inner-product argument* from Section 3.3.4 and are also more efficient in practice. Let $1_{\mathbb{G}}$ denote the identity in a group $\mathbb{G}$.

Vectors. Bolded, lower-case symbols such as $\mathbf{a} = [a_0, \ldots, a_{n-1}] \in \mathbb{Z}_p^n$ typically denote vectors of field elements. Bolded, upper-case symbols such as $\mathbf{A} = [A_1, \ldots, A_m] \in \mathbb{G}^m$ typically denote vectors of group elements. $|\mathbf{A}|$ denotes the size of the vector $\mathbf{A}$. $\mathbf{A}^x = [A_1^x, \ldots, A_m^x], x \in \mathbb{Z}_p$, $\mathbf{A} \circ \mathbf{B} = [A_1 B_1, A_2 B_2, \ldots, A_m B_m]$, and $\langle \mathbf{A}, \mathbf{B} \rangle = \prod_{j=1}^m e(A_j, B_j)$ denotes a *pairing product*. Let $\mathbf{A}_L = [A_1, \ldots, A_{m/2}]$ and $\mathbf{A}_R = [A_{m/2+1}, \ldots, A_m]$ denote the left and right halves of $\mathbf{A}$. Let $\mathbf{A}||1_{\mathbb{G}}$ denote a vector of size $2|\mathbf{A}|$ that "extends" $\mathbf{A}$ to the right with the identity of $\mathbb{G}$. (Similarly, $1_{\mathbb{G}}||\mathbf{A}$ "extends" $\mathbf{A}$ to the left.)

3.3.1 Multilinear extension (MLE) of a vector

Let $n = 2^\ell$ and $\mathbf{x} = (x_\ell, \ldots, x_1)$. A vector $\mathbf{a} = [a_0, \ldots, a_{n-1}] \in \mathbb{Z}_p^n$ can be represented as a *multilinear extension* polynomial $f : \mathbb{Z}_p^\ell \to \mathbb{Z}_p$ which maps each position i to a_i:

$$f(\mathbf{i}) = f(i_\ell, \ldots, i_2, i_1) = a_i, \forall i \in [0, n) \tag{3.2}$$

For example, the MLE of $\mathbf{a} = [5, 2, 8, 3]$ is $f(x_2, x_1)$ defined as:

$$5(1 - x_2)(1 - x_1) + 2(1 - x_2)x_1 + 8x_2(1 - x_1) + 3x_2x_1 \tag{3.3}$$

In general, the *unique* multilinear extension f of $\mathbf{a}$ is:

$$f(\mathbf{x}) = f(x_\ell, \ldots, x_1) = \sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(x_\ell, \ldots, x_1) = \sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{x}) \tag{3.4}$$

where $\mathcal{S}_{j,\ell}, j \in [0, 2^\ell)$ are *selector multinomials* defined as:

$$\mathcal{S}_{j,\ell}(\mathbf{x}) = \prod_{k=1}^{\ell} \mathsf{sel}_{j_k}(x_k), \text{s.t. } \mathsf{sel}_{j_k}(x_k) = \begin{cases} x_k, & \text{if } j_k = 1 \\ 1 - x_k, & \text{if } j_k = 0 \end{cases}, \tag{3.5}$$

with $S_{0,0}(\mathbf{x}) = 1$. In our example from Eq. (3.3), we have $\ell = 2$ and so: $\mathcal{S}_{0,2}(\mathbf{x}) = (1 - x_2)(1 - x_1)$, $\mathcal{S}_{1,2}(\mathbf{x}) = (1 - x_2)x_1$, $\mathcal{S}_{2,2}(\mathbf{x}) = x_2(1 - x_1)$ and $\mathcal{S}_{3,2}(\mathbf{x}) = x_2x_1$. We often refer to sel_{j_k} as a *selector monomial*. Importantly, note that:

$$\mathcal{S}_{j,\ell}(i_\ell, \ldots, i_1) = \mathcal{S}_{j,\ell}(\mathbf{i}) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \forall i \in [0, 2^\ell) \tag{3.6}$$

By these properties above, we can see why Eq. (3.2) holds for any i:

$$f(\mathbf{i}) = \sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{i}) = a_i \mathcal{S}_{i,\ell}(\mathbf{i}) + \sum_{j=1, j\neq i}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{i}) = a_i \cdot 1 + 0$$

In other words, an MLE f acts as a "multiplexer", choosing the right a_i based on the input position i, given as $\mathbf{i}$ in binary.

MLE decomposition. An MLE of size $n = 2^\ell$ can be *decomposed* into two MLEs of size $n/2$ [163]. For example, split $\mathbf{a}$ from Eq. (3.3) into its left and right halves $\mathbf{a}_0 = [5, 2]$ and $\mathbf{a}_1 = [8, 3]$, with MLEs $f_0 = 5(1 - x_1) + 2x_1$ and $f_1 = 8(1 - x_1) + 3x_1$, respectively. Then, observe that the MLE f for $\mathbf{a}$ is a combination of f_0 and f_1: i.e., $f = (1 - x_2)f_0 + x_2 f_1$. In general, the MLE f of any $\mathbf{a}$ decomposes as:

$$f(\mathbf{x}) = (1 - x_\ell)f_0(x_{\ell-1}, \ldots, x_1) + x_\ell f_1(x_{\ell-1}, \ldots, x_1) \tag{3.7}$$

Note that for $\mathbf{a} = [a_0, a_1]$ of size 2, the MLEs f_0, f_1 are *trivial* (i.e., of size 1) and are simply set to a_0 and a_1, respectively. We use $f_{b_\ell b_{\ell-1} \ldots b_k}$ to denote the MLE of the $\mathbf{a}_{b_\ell b_{\ell-1} \ldots b_k}$ subvector, which is a subvector of all a_i's with $i_\ell = b_\ell, i_{\ell-1} = b_{\ell-1}, \ldots, i_k = b_k$. For example, in a vector $\mathbf{a} = [a_0, \ldots, a_7]$, f_{01} is the MLE of $\mathbf{a}_{01}$, which contains all (three bit) positions i whose first two bits are 01: i.e., $\mathbf{a}_{01} = [a_2, a_3]$ because, in binary, 2 and 3 are 010 and 011, respectively.

3.3.2 PST commitments to MLEs

Papamanthou, Shi and Tamassia [108] extend Kate-Zaverucha-Goldberg (KZG) univariate polynomial commitments [82] to multivariate ones. We refer to their scheme as PST and restrict its use to multilinear extensions, introduced above.

Commitments. PST works over a bilinear group obtained via BilGen. The PST commitment to a multilinear extension f for a vector $\mathbf{a}$ of size $n = 2^\ell$ is a single group element in

$\mathbb{G}_1$:

$$\mathsf{pst}(f) = g_1^{f(s_\ell,\dots,s_1)} = g_1^{\sum_{j=0}^{n-1} a_j \mathcal{S}_{j,\ell}(\mathbf{s})} = \prod_{j=0}^{n-1} \left(g_1^{\mathcal{S}_{j,\ell}(\mathbf{s})}\right)^{a_j} \tag{3.8}$$

Here, $\mathbf{s} = (s_\ell, \dots, s_1)$ are *trapdoors* generated via a *trusted setup* that outputs n-sized *public parameters*: $g_1^{\mathcal{S}_{j,\ell}(\mathbf{s})} = g_1^{\mathcal{S}_{j,\ell}(s_\ell,\dots,s_1)}, \forall j \in [0, 2^\ell)$. Importantly, the setup discards $\mathbf{s}$, since knowledge of it directly breaks PST's security [107]. We stress that $\mathsf{pst}(f)$ can be computed without knowing $\mathbf{s}$, as per Eq. (3.8). Lastly, PST commitments are *homomorphic*, with $\mathsf{pst}(f + f') = \mathsf{pst}(f)\mathsf{pst}(f')$ for any MLEs f, f'.

Evaluation proofs. Papamanthou, Shi and Tamassia give a way to prove evaluations $f(\mathbf{i})$ against $\mathsf{pst}(f)$ [108], where $\mathbf{i}$ is the binary representation of $i \in [0, n)$. Their key observation, which we refer to as the *PST decomposition*, is that:

$$f(\mathbf{i}) = z \Leftrightarrow \exists q_j\text{'s}, f(\mathbf{x}) - z = \sum_{j \in [\ell]} q_j(x_{j-1}, \dots, x_1) \cdot (x_j - i_j) \tag{3.9}$$

This yields a *PST evaluation proof* for $f(\mathbf{i}) = z$ consisting of commitments $w_j = g_1^{q_j(\mathbf{s})}$ to the *quotient polynomials* q_j. To compute the q_j's, the prover first divides f by $x_\ell - i_\ell$, obtaining q_ℓ and a remainder r_ℓ. Then, the prover continues recursively on the remainder r_ℓ, which no longer has variable x_ℓ. Specifically, the prover divides r_ℓ by $x_{\ell-1} - i_{\ell-1}$, obtaining $q_{\ell-1}$ and $r_{\ell-1}$. And so on, until he obtains the last quotient q_1 with remainder $r_1 = f(\mathbf{i})$ (see Fig. 3.2 and [107, Lemma 1]). Overall, this takes $T(n) = O(n) + T(n/2) = O(n)$ time, including the time to commit to the q_j's.

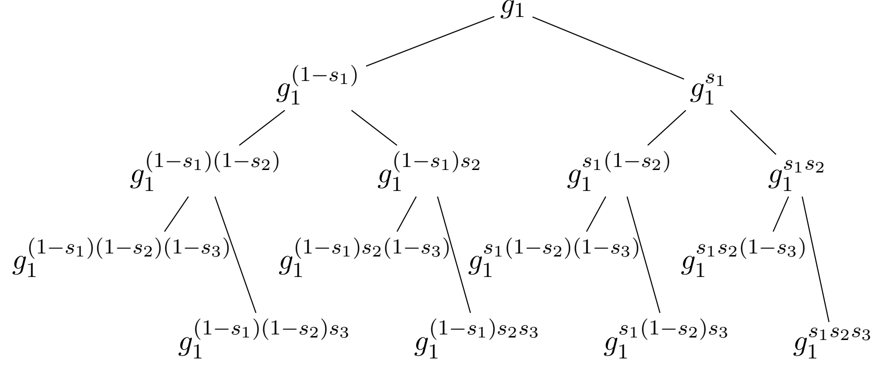


Figure 3.1: PST (and Hyperproofs) public parameters. The uth path in this tree is actually the update key upk_u from Eq. (3.16).

Note that the q_j's are actually MLEs of size $n/2, n/4, \ldots, 1$. As a result, PST's actual public parameters are $g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0, \ell], \forall j \in [0, 2^k)$, so as to also be able to commit to these quotient MLEs. Lastly, the parameters form a tree (see Fig. 3.1) and are thus of size $2n - 1$ $\mathbb{G}_1$ elements.

A verifier who has the commitment $\mathsf{pst}(f)$, the claimed evaluation $(i, f(\mathbf{i}) = z)$ and a logarithmic-sized, publicly-known *verification key* $g_2^{s_j}, \forall j \in [\ell]$ can verify the proof using $\ell + 1$ pairings:

$$e(\mathsf{pst}(f)/g_1^z, g_2) = \prod_{j \in [\ell]} e(w_j, g_2^{s_j - i_j}) \tag{3.10}$$

The check above ensures Eq. (3.9) holds when $\mathbf{x} = \mathbf{s}$, which is sufficient for security since $\mathbf{s}$ is random and secret. In constructing our VC, we prove a stronger notion of security for PST commitments (see Section 3.7).

3.3.3 Vector Commitments (VCs)

We formalize VCs below, similar to Catalano and Fiore [46].

$\mathsf{PST.Prove}(f, \ell, \mathbf{i} = (i_\ell, \ldots, i_1)) \to \pi_i$:

1. If $\ell = 0$ (i.e., f is a constant), return $\varnothing$.
2. Otherwise, divide f by $x_\ell - i_\ell$, obtaining quotient $q_\ell(x_{\ell-1}, \ldots, x_1)$ and remainder $r_\ell(x_{\ell-1}, \ldots, x_1)$ such that $f = q_\ell \cdot (x_\ell - i_\ell) + r_\ell$.
3. Return $\left(g_1^{q_\ell(\mathbf{s})}, \mathsf{PST.Prove}(r_\ell, \ell - 1, (i_{\ell-1}, \ldots, i_1))\right)$

Figure 3.2: $O(n)$-time algorithm for computing a single PST evaluation proof π_i for $f(\mathbf{i})$ w.r.t. an MLE f of size $n = 2^\ell$.

Definition 3.3.1 (VC). *A VC scheme is a set of* PPT *algorithms:*

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$: Given security parameter λ and maximum vector size n, outputs randomly-generated public parameters pp.

$\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}$: Outputs digest C of $\mathbf{a} = [a_0, \ldots, a_{n-1}] \in \mathbb{Z}_p^n$.

$\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \to \pi_i$: Outputs a proof π_i for position i in $\mathbf{a}$.

$\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \to (\pi_0, \ldots, \pi_{n-1})$: Outputs all proofs π_i for $\mathbf{a}$.

$\mathsf{Agg}_{\mathsf{pp}}(I, (a_i, \pi_i)_{i \in I}) \to \pi_I$: Combines individual proofs π_i for values a_i into an aggregated proof π_I.

$\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i)_{i \in I}, \pi_I) \to \{0, 1\}$: Verifies proof π_I that each position $i \in I$ has value a_i against digest C.

$\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}) \to \mathsf{C}'$: Updates digest C to C' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i) \to \pi_i'$: Updates proof π_i to π_i' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi_0', \ldots, \pi_{n-1}')$: Updates all proofs π_i to π_i' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

Observations: For simplicity, we give our algorithms oracle access to the public parameters pp of the scheme. This way, each algorithm can easily access the subset of the parameters it needs.

We formalize $\mathsf{OpenAll}$ and $\mathsf{UpdAllProofs}$ since, in some VCs, these algorithms are faster than n calls to Open and $\mathsf{UpdProof}$, respectively. In this sense, we stress that the $\mathsf{UpdAllProofs}$ algorithm can work in sublinear time, since it does not necessarily need to read all input or write all output (e.g., in Merkle trees, $\mathsf{UpdAllProofs}$ only reads $\log n$ sibling hashes and overwrites another $\log n$ hashes).

Correctness and soundness. We define VC correctness in Definition 3.9.1 and VC soundness in Definition 3.9.2.

3.3.4 Inner Product Arguments (IPA)

Let CM denote a commitment scheme by Abe et al. [2] for vectors $\mathbf{A}, \mathbf{B} \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ and their *pairing product* $Z = \langle \mathbf{A}, \mathbf{B} \rangle = \prod_{i=1}^m e(A_i, B_i)$. CM uses a randomly-generated *commitment key* $\mathbf{ck} = (\mathbf{v}, \mathbf{w}) \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ to commit to $\mathbf{A}, \mathbf{B}$ and Z as:

$$\mathbf{C} = \mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}, Z) = (\langle \mathbf{A}, \mathbf{v} \rangle,\ \langle \mathbf{w}, \mathbf{B} \rangle,\ Z) \stackrel{\mathsf{def}}{=} (C_1, C_2, C_3) \tag{3.11}$$

This commitment scheme is not hiding but is binding under *Symmetric-eXternal Diffie-Hellman (SXDH)* (see Assumption 3.8.1) [2, 3].

Bünz et al. [34] give a non-interactive *inner-product argument (IPA)* where a *prover* convinces a *verifier*, that the prover *knows* how to open an Abe et al. commitment $\mathbf{C}$ to

$(\mathbf{A}, \mathbf{B}, \langle \mathbf{A}, \mathbf{B} \rangle)$; i.e. they give an argument for the language:

$$\mathcal{L}^m_{\mathsf{IPA}} = \{(\mathbf{ck}, \mathbf{C}) \mid \exists \mathbf{A} \in \mathbb{G}_1^m, \mathbf{B} \in \mathbb{G}_2^m, \text{ s.t. } \mathbf{C} = \mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}, \langle \mathbf{A}, \mathbf{B} \rangle)\}$$

We abstract Bünz et al.'s [34] non-interactive argument for $\mathcal{L}_{\mathsf{IPA}}$ as three algorithms:

$\underline{\mathcal{G}_{\mathsf{IPA}}(1^\lambda, m) \to (PK, VK)}$: Returns $PK = VK = \langle \mathsf{BilGen}(1^\lambda), \mathbf{ck} = (\mathbf{v} \in_R \mathbb{G}_2^m, \mathbf{w} \in_R \mathbb{G}_1^m) \rangle$

$\underline{\mathcal{P}_{\mathsf{IPA}}(PK, \mathbf{A}, \mathbf{B}) \to \pi}$: Returns a proof π that $\mathbf{C} = \mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}, \langle \mathbf{A}, \mathbf{B} \rangle)$

$\underline{\mathcal{V}_{\mathsf{IPA}}(VK, \mathbf{C}, \pi) \to \{0, 1\}}$: Verifies proof π that $\mathbf{C} = \mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}, \langle \mathbf{A}, \mathbf{B} \rangle)$

IPA complexity. $\mathcal{P}_{\mathsf{IPA}}$ takes $O(m)$ time, $\mathcal{V}_{\mathsf{IPA}}$ takes $O(\log m)$ time and the proof size is $|\pi| = O(\log m)$ (see Section 3.8).

3.4 Hyperproofs

In this section, we intuitively explain how Hyperproofs work, often referring to a *prover* who computes the vector's *digest*, as well as proofs, and to a *verifier* who verifies proofs against this digest. Without loss of generality, our discussion will assume vectors of size exactly $n = 2^\ell$. Hyperproofs represents a vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]$ as a multilinear extension (MLE):

$$f(\mathbf{x}) = \sum_{i=0}^{n-1} a_i \mathcal{S}_{i,\ell}(\mathbf{x}),$$

where $\mathcal{S}_{i,\ell}$ are selector multinomials as per Eq. (3.5). The *digest* of the vector $\mathbf{a}$ is a Papamanthou-Shi-Tamassia (PST) commitment to f:

$$\mathsf{pst}(f) = g_1^{f(\mathbf{s})},$$

where $\mathbf{s}$ is the PST trapdoor (see Section 3.3.2). Thus, our public parameters are the same as PST's parameters depicted in Fig. 3.1.

3.4.1 Multilinear trees (MLTs)

A Hyperproof for position i is just a PST evaluation proof (see Section 3.3.2) for $f(\mathbf{i})$. Unfortunately, if one uses the $\mathsf{PST.Prove}$ algorithm from Fig. 3.2 to compute *all* PST evaluation proofs, this takes $O(n^2)$ time. Below, we show how to compute all n proofs faster, in $O(n \log n)$ time, by avoiding unnecessary computations (see Fig. 3.4).

Denote the proof for $f(\mathbf{i})$ as $\pi_i = (\pi_{i,\ell}, \ldots, \pi_{i,1})$. Next, observe that if we compute all proofs π_i via n calls to:

$$\mathsf{PST.Prove}(f, \ell, (i_\ell, \ldots, i_1)), \forall i \in [n],$$

they actually all have the same first quotient q_ℓ committed in $\pi_{i,\ell}$! This is because all n $\mathsf{PST.Prove}$ calls initially divide f by $x_\ell - i_\ell$, which actually yields the same quotient, independent of i_ℓ. To see this, recall the MLE decomposition from Eq. (3.7) and reorganize

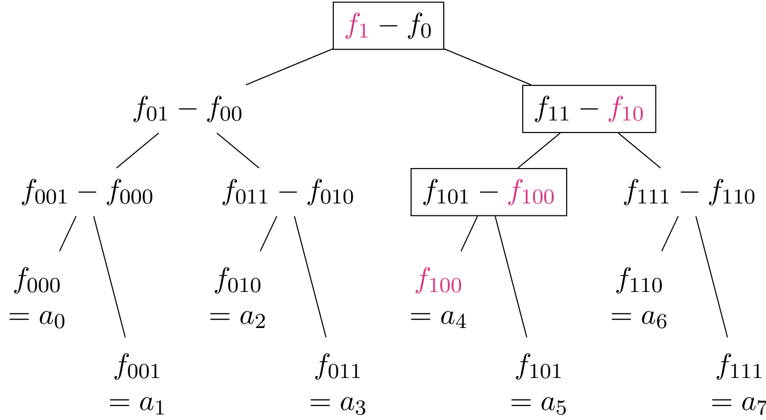


Figure 3.3: A multilinear tree (MLT) of size 8. Recall from Section 3.3 that $f_{b_\ell b_{\ell-1} \ldots b_k}$ denotes the MLE of $\mathbf{a}_{b_\ell b_{\ell-1} \ldots b_k}$. Each node stores a PST commitment to the depicted MLE: e.g., root stores $\mathsf{pst}(f_1 - f_0)$, not $f_1 - f_0$. The proof for a_i consists of all commitments along a_i's path to the root (e.g., for a_4, the boxed nodes). Sibling leaves $[a_{2j}, a_{2j+1}]$ have the same proof. If, say, a_4 changes, all pink-colored MLEs change, and all boxed commitments must be updated.

it in two ways as:

$$f = (1 - x_\ell) \cdot f_0 + x_\ell \cdot f_1 \Leftrightarrow$$

$$f = (f_1 - f_0) \cdot (x_\ell - 1) + f_1 \tag{3.12}$$

$$= (f_1 - f_0) \cdot x_\ell + f_0 \,, \tag{3.13}$$

where f_0 is the MLE for the left half $\mathbf{a}_0$ of $\mathbf{a}$ and f_1 is the MLE for the right half $\mathbf{a}_1$ (recall from Section 3.3). Since both divisions yield the same $q_\ell = f_1 - f_0$ quotient, all π_i's share the same $\pi_{i,\ell}$ commitment to q_ℓ! We depict this q_ℓ as the root of a *multilinear tree (MLT)* in Fig. 3.3.

Next, recall that each one of the n $\mathsf{PST.Prove}$ calls recurses on its remainder, which was either f_0 or f_1 (as per Eqs. (3.12) and (3.13)). Specifically, the first $n/2$ calls for $i \in [0, n/2)$ (i.e., $i_\ell = 0$) recurse on $\mathsf{PST.Prove}(f_0, \ell - 1, (i_{\ell-1}, \ldots, i_1))$, and the other $n/2$

$\mathsf{MLT.Compute}(f, \ell) \to [t_1, \ldots, t_{2^\ell - 1}]$:

1. If $\ell = 0$ (i.e., f is a constant), return $\varnothing$.
2. Otherwise, $\forall b \in \{0, 1\}$, divide f by $x_\ell - b$, obtaining quotient $f_1 - f_0$ and remainder f_b such that $f = (f_1 - f_0) \cdot (x_\ell - b) + f_b$.
3. Return $\left(g_1^{(f_1 - f_0)(\mathbf{s})}, \mathsf{MLT.Compute}(f_0, \ell - 1), \mathsf{MLT.Compute}(f_1, \ell - 1)\right)$

Figure 3.4: Computes an MLT in $O(n \log n)$ time consisting of PST evaluation proofs for *all* $f(\mathbf{i})$ w.r.t. an MLE f of $\mathbf{a}$ of size $n = 2^\ell$. In contrast, n naive calls to $\mathsf{PST.Prove}$ would take $O(n^2)$. Recall that f_0 and f_1 are MLEs for the left and right halves of $\mathbf{a}$. Returns the tree stored in preorder in an array.

calls for $i \in [n/2+1, n)$ (i.e., $i_\ell = 1$) recurse on $\mathsf{PST.Prove}(f_1, \ell-1, (i_{\ell-1}, \ldots, i_1))$. But by the same argument above, each group of $n/2$ calls returns the same first quotient commitment. For example, for the first group, we have quotient $f_{01} - f_{00}$:

$$f_0 = (f_{01} - f_{00})(x_{\ell-1} - 1) + f_{01} \tag{3.14}$$

$$= (f_{01} - f_{00})x_{\ell-1} + f_{00}\,, \tag{3.15}$$

Similarly, for the second group, the quotient will be $f_{11} - f_{10}$. Both quotients are depicted as the children of the root in Fig. 3.3. Continuing recursively in this fashion yields our *multilinear tree (MLT)* from Fig. 3.3. We describe the algorithm for computing it in Fig. 3.4 and we argue correctness of MLT proofs in Section 3.9.

3.4.2 Updates and homomorphism

Updating digests and MLTs. Suppose a_4 changes by δ in our MLT from Fig. 3.3. Then, by Eq. (3.4), we know that $\mathbf{a}$'s MLE will change to:

$$f' = f + x_3(1 - x_2)(1 - x_1)\delta = f + \mathcal{S}_{4,3}(\mathbf{x})\delta$$

But what about the MLT? The following highlighted MLEs from Fig. 3.3 will be updated to:

$$f'_{100} = f_{100} + \delta$$

$$f'_{10} = f_{10} + (1 - x_1)\delta$$

$$f'_1 = f_1 + (1 - x_2)(1 - x_1)\delta$$

$$f' = f + x_3(1 - x_2)(1 - x_1)\delta$$

These MLEs changing affect the MLEs along a_4's path. For example, the root MLE $f_1 - f_0$ also changes by the same amount as f_1: i.e., by $+(1 - x_2)(1 - x_1)\delta$. Furthermore, their corresponding commitments are easy to update via the PST homomorphism. For example, the new root will be $\mathsf{pst}(f_1 - f_0) \cdot g_1^{(1-s_2)(1-s_1)\delta}$. However, note that updating commitments requires knowing $g_1^{(1-s_2)(1-s_1)}$, which is referred to as an *update key*. We delve into this next.

Update keys. Recall that $\mathcal{S}_{u,k}(\mathbf{x})$ is the selector multinomial for position $u \in [0, 2^k)$ in an MLE of size 2^k (see Eq. (3.5)). However, to easily reason about updates, it is useful to define $\mathcal{S}_{u,k}$ even when $u \geq 2^k$ as $\mathcal{S}_{u,k} = \mathcal{S}_{u \bmod 2^k, k}$. As explained above, updating the MLT after a_u changes by δ requires some auxiliary information referred to as an *update key* for position u. This consists of commitments to all selector multinomials for u in MLEs of size $1, 2, \ldots, 2^\ell$:

$$\mathsf{upk}_u = \left\{ g_1^{\mathcal{S}_{u,k}(\mathbf{s})} : k \in [0, \ell] \right\} = \left\{ \mathsf{upk}_{u,k} : k \in [0, \ell] \right\} \tag{3.16}$$

Recall that $\mathcal{S}_{u,0}(\mathbf{x}) = 1$, so that $\mathsf{upk}_{u,0} = g_1, \forall u \in [0, 2^\ell)$. Then, the MLT commitments $(w_{u,\ell}, \ldots, w_{u,1})$ along u's path are updated as:

$$w'_{u,k} = w_{u,k} \cdot (\mathsf{upk}_{u,k-1})^\delta = w_{u,k} \cdot (g_1^{\mathcal{S}_{u,k-1}(\mathbf{s})})^\delta, \forall k \in [\ell] \tag{3.17}$$

Note that this implies that any proof $\pi_i = (w_{i,\ell}, \ldots, w_{i,1})$ can be updated after a change at u: one simply has to identify the "intersection" of u's proof with i's proof and apply the update as above, as if updating a pruned MLT consisting of just π_i. More formally, suppose i and u have the same t most significant bits (i.e., $i_k = u_k, \forall k \in \{\ell, \ell-1, \ldots, \ell-t+1\}$). Then, the updated proof π'_i is initially set to π_i and (partially) updated as:

$$w'_{i,k} = w_{i,k} \cdot (\mathsf{upk}_{u,k-1})^\delta, \forall k \in \{\ell, \ldots, \ell-t\}, 1 \le k \le \ell \tag{3.18}$$

The digest updates more simply as:

$$\mathsf{pst}(f') = \mathsf{pst}(f) \cdot g_1^{\mathcal{S}_{u,\ell}(\mathbf{s})} = \mathsf{pst}(f) \cdot \left(\mathsf{upk}_{u,\ell}\right)^\delta \tag{3.19}$$

Lastly, we note that the update keys actually coincide with our public parameters (see Fig. 3.1).

MLTs are homomorphic. Since our multilinear tree stores an MLE commitment at each node, we observe that the MLT itself is *homomorphic*: the MLT for $\mathbf{a} + \mathbf{b}$ can be obtained by "node-by-node multiplying" $\mathbf{a}$'s MLT with $\mathbf{b}$'s MLT. In other words, every node w in the new MLT is the product of the nodes w in the MLTs for $\mathbf{a}$ and $\mathbf{b}$. Specif-

ically, $\mathsf{pst}(f''_w) = \mathsf{pst}(f_w + f'_w) = \mathsf{pst}(f_w)\mathsf{pst}(f'_w)$, where f_w, f'_w, f''_w denote the MLE stored at node w in the MLT for $\mathbf{a}, \mathbf{b}$ and $\mathbf{a} + \mathbf{b}$, respectively. This enables our unstealability construction from Section 3.4.4 and has other applications to authenticating data in the *streaming* setting [109].

3.4.3 Aggregating proofs

Recall that a proof $(w_1, \ldots, w_\ell)$ for a_i in the vector $\mathbf{a}$ of size $n = 2^\ell$ is just an ℓ-sized PST evaluation proof (see Section 3.3.2) and verifies as:

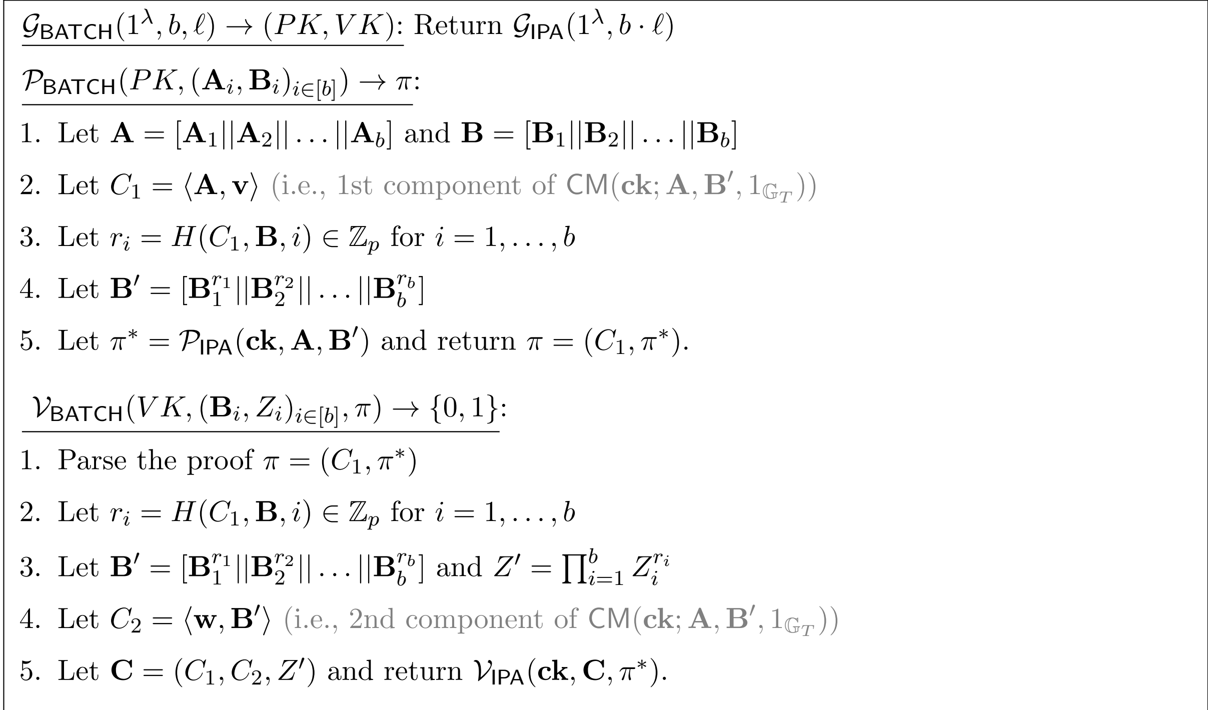
$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{k=1}^{\ell} e(w_k, g_2^{s_k - i_k})\,, \tag{3.20}$$

where C is the digest and $(g_2^{s_k - i_k})_{k \in [\ell]}$ is position i's public *verification key.*

Warm-up: Compressing proofs. It is useful to first discuss compressing a size-ℓ proof for a_i to size $\log \ell$ via the IPA from Section 3.3.4. For this, we let $\mathbf{A} = [w_1 \ldots w_\ell]$, $\mathbf{B} = [g_2^{s_1 - i_1} \ldots g_2^{s_\ell - i_\ell}]$, $Z = e(\mathsf{C}/g_1^{a_i}, g_2)$ and prove that $(Z, \mathbf{B})$ is in the following language:

$$\mathcal{L}^{\ell}_{\mathsf{PROD}} = \left\{ Z \in \mathbb{G}_T, \mathbf{B} \in \mathbb{G}_2^{\ell} \;\middle|\; \exists \mathbf{A} \in \mathbb{G}_1^{\ell}, Z = \langle \mathbf{A}, \mathbf{B} \rangle \right\} \tag{3.21}$$

Next, we can use the IPA from Section 3.3.4. Specifically, assume our $\mathcal{L}_{\mathsf{PROD}}$ prover and verifier share a commitment key $\mathbf{ck} = (\mathbf{v}, \mathbf{w})$. First, the prover gives $C_1 = \langle \mathbf{A}, \mathbf{v} \rangle$ to the verifier. Second, the verifier computes $C_2 = \langle \mathbf{w}, \mathbf{B} \rangle$ and lets $C_3 = Z$. Thus, the verifier now has a commitment $\mathbf{C} = (C_1, C_2, C_3)$ to $\mathbf{A}$,$\mathbf{B}$ and Z. Third, the prover simply runs $\mathcal{P}_{\mathsf{IPA}}$



$\underline{\mathcal{G}_{\mathsf{BATCH}}(1^\lambda, b, \ell) \to (PK, VK)}$: Return $\mathcal{G}_{\mathsf{IPA}}(1^\lambda, b \cdot \ell)$

$\underline{\mathcal{P}_{\mathsf{BATCH}}(PK, (\mathbf{A}_i, \mathbf{B}_i)_{i\in[b]}) \to \pi}$:

1. Let $\mathbf{A} = [\mathbf{A}_1 || \mathbf{A}_2 || \dots || \mathbf{A}_b]$ and $\mathbf{B} = [\mathbf{B}_1 || \mathbf{B}_2 || \dots || \mathbf{B}_b]$
2. Let $C_1 = \langle \mathbf{A}, \mathbf{v} \rangle$ (i.e., 1st component of $\mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}', 1_{\mathbb{G}_T})$)
3. Let $r_i = H(C_1, \mathbf{B}, i) \in \mathbb{Z}_p$ for $i = 1, \dots, b$
4. Let $\mathbf{B}' = [\mathbf{B}_1^{r_1} || \mathbf{B}_2^{r_2} || \dots || \mathbf{B}_b^{r_b}]$
5. Let $\pi^* = \mathcal{P}_{\mathsf{IPA}}(\mathbf{ck}, \mathbf{A}, \mathbf{B}')$ and return $\pi = (C_1, \pi^*)$.

$\underline{\mathcal{V}_{\mathsf{BATCH}}(VK, (\mathbf{B}_i, Z_i)_{i\in[b]}, \pi) \to \{0,1\}}$:

1. Parse the proof $\pi = (C_1, \pi^*)$
2. Let $r_i = H(C_1, \mathbf{B}, i) \in \mathbb{Z}_p$ for $i = 1, \dots, b$
3. Let $\mathbf{B}' = [\mathbf{B}_1^{r_1} || \mathbf{B}_2^{r_2} || \dots || \mathbf{B}_b^{r_b}]$ and $Z' = \prod_{i=1}^{b} Z_i^{r_i}$
4. Let $C_2 = \langle \mathbf{w}, \mathbf{B}' \rangle$ (i.e., 2nd component of $\mathsf{CM}(\mathbf{ck}; \mathbf{A}, \mathbf{B}', 1_{\mathbb{G}_T})$)
5. Let $\mathbf{C} = (C_1, C_2, Z')$ and return $\mathcal{V}_{\mathsf{IPA}}(\mathbf{ck}, \mathbf{C}, \pi^*)$.

Figure 3.5: Our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ used to aggregate Hyperproofs. H is a random oracle and $(\mathcal{G}_{\mathsf{IPA}}, \mathcal{P}_{\mathsf{IPA}}, \mathcal{V}_{\mathsf{IPA}})$ is the Bünz et al. IPA from Section 3.3.4.

from Section 3.3.4 and convinces the verifier that the committed values satisfy $Z = \langle \mathbf{A}, \mathbf{B} \rangle$ and thus that the Hyperproof verifies as per Eq. (3.20).

Aggregating proofs. Next, we observe that aggregating many proofs $(\pi_1, \dots, \pi_b)$, each for a position p_i in $\mathbf{a}$, reduces to proving membership in $\mathcal{L}^{\ell}_{\mathsf{PROD}}$ *for each* $(Z_i, \mathbf{B}_i)$, where $Z_i = e(\mathsf{C}/g_1^{a_{p_i}}, g_2)$ and $\mathbf{B}_i$ is position p_i's verification key. But doing this naively would result in a large, $O(b \log \ell)$ aggregated proof size. Instead, we seek a more succinct argument for the following new language:

$$\mathcal{L}^{b,\ell}_{\mathsf{BATCH}} = \left\{ (Z_i \in \mathbb{G}_T, \mathbf{B}_i \in \mathbb{G}_2^{\ell})_{i\in[b]} \;\middle|\; ((Z_i, \mathbf{B}_i) \in \mathcal{L}^{\ell}_{\mathsf{PROD}})_{i\in[b]} \right\} \tag{3.22}$$

$$= \left\{ (Z_i \in \mathbb{G}_T, \mathbf{B}_i \in \mathbb{G}_2^{\ell})_{i\in[b]} \;\middle|\; (\exists \mathbf{A}_i \in \mathbb{G}_1^{\ell}, Z_i = \langle \mathbf{A}_i, \mathbf{B}_i \rangle)_{i\in[b]} \right\}$$

In other words, membership in $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ guarantees that $\forall i \in [b], \exists \mathbf{A}_i$:

$$Z_i = \prod_{j=1}^{\ell} e(A_{i,j}, B_{i,j}), \tag{3.23}$$

where $A_{i,j}$ is the jth entry of $\mathbf{A}_i$. Note that we cannot use the TIPP argument from [34] to prove membership in $\mathcal{L}_{\mathsf{BATCH}}$, since it can only prove that $\forall i, Z_i = e(X_i, Y_i)$, where $(X_i, Y_i) \in \mathbb{G}_1 \times \mathbb{G}_2$. Instead, we design a new argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ (see Fig. 3.5) which uses a random linear combination to combine the ℓ-sized equations from above into a single $b\ell$-sized one:

$$\prod_{i=1}^{b} Z_i^{r_i} = \prod_{i=1}^{b} \left(\prod_{j=1}^{\ell} e(A_{i,j}, B_{i,j}) \right)^{r_i} \tag{3.24}$$

It is well known that, if the r_i's are uniformly random, verifying the combined equation above is sufficient (see Lemma 3.8.1). As a result, our argument for $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ uses the IPA from Section 3.3.4 on this combined equation in a black-box manner. (This is similar to the previous $\mathcal{L}^{\ell}_{\mathsf{PROD}}$ argument, except it involves larger vectors and randomization.) We give a precise description of its $(\mathcal{G}_{\mathsf{BATCH}}, \mathcal{P}_{\mathsf{BATCH}}, \mathcal{V}_{\mathsf{BATCH}})$ algorithms in Fig. 3.5, show how it fits in our VC construction in Fig. 3.6, and prove security in Section 3.8.

$\underline{\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}}$: Let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$. Let $\mathbf{s} = (s_1, \ldots, s_\ell) \in_R \mathbb{Z}_p^\ell$, where $n = 2^\ell$.

Let pp consist of

- $\mathsf{pst}(\mathcal{S}_{j,k}) = g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0, \ell], \forall j \in [0, 2^k)$;
- $g_2^{s_k}, \forall k \in [\ell]$;
- $(PK, VK) \leftarrow \mathcal{G}_{\mathsf{BATCH}}(1^\lambda, b, \ell)$.

We refer to $(g_2^{s_k - i_k})_{k \in [\ell]}$ as position i's *verification key.*

$\underline{\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}}$: Let $\mathsf{C} = \mathsf{pst}(f) = g_1^{f(\mathbf{s})} = g_1^{f(s_1, \ldots, s_\ell)}$, where f is $\mathbf{a}$'s MLE.

$\underline{\mathsf{OpenAll}_{\mathsf{pp}}(\boldsymbol{a}) \to (\pi_0, \ldots, \pi_{n-1})}$: Return the MLT as per Section 3.4.1.

$\underline{\mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}) \to \pi_i}$: Compute only the ith path in the MLT and return it.

$\underline{\mathsf{Agg}_{\mathsf{pp}}(I, \{a_i, \pi_i\}_{i \in I}) \to \pi_I}$: Let $m = |I|$ and let $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_m$ denote proofs $(\pi_i)_{i \in I}$, ordered by i, and $\mathbf{B}_1, \ldots, \mathbf{B}_m$ denote their corresponding verification keys. Return $\mathcal{P}_{\mathsf{BATCH}}(PK, (A_k, B_k)_{k \in [m]})$.

$\underline{\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, \{a_i\}_{i \in I}, \pi_I) \to \{0, 1\}}$: If $I = \{i\}$, parse $\pi_I = (w_1, \ldots, w_\ell)$ and ensure that

$$e(\mathsf{C}/g_1^{a_i}, g_2) = \prod_{j=1}^{\ell} e(w_j, g_2^{s_j - i_j}) \,.$$

Otherwise, let $m = |I|$ and $\mathbf{B}_1, \ldots, \mathbf{B}_m$ denote the verification keys for the proofs, ordered by their position i. Let $Z_1, Z_2, \ldots, Z_m$ be all the $e(\mathsf{C}/g_1^{a_i}, g_2)$'s, also ordered by i. Return $\mathcal{V}_{\mathsf{BATCH}}(VK, (B_k, Z_k)_{k \in [m]}, \pi_I)$.

$\underline{\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}) \to \mathsf{C}'}$: Let $\mathsf{C}' = \mathsf{C} \cdot (g_1^{\mathcal{S}_{u,\ell}(\mathbf{s})})^\delta$.

$\underline{\mathsf{UpdProof}_{\mathsf{pp}}(u, \delta, \pi_i) \to \pi_i'}$: Update via $\mathsf{UpdAllProofs}$ (see below) as if π_i was a pruned, single-path MLT (see Eq. (3.18)).

$\underline{\mathsf{UpdAllProofs}_{\mathsf{pp}}(u, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi_0', \ldots, \pi_{n-1}')}$: Assume u's MLT path is $(w_1, \ldots, w_\ell)$. Update this path as $w_k' = w_k \cdot (\mathsf{upk}_{u,k-1})^\delta$ (for $k = 1, \ldots, \ell$) as per Eq. (3.17).

Figure 3.6: Algorithms for Hyperproofs, implicitly parameterized by the max number of proofs b that can be aggregated into a single proof.

Aggregation time and proof size. It is easy to see from Fig. 3.5 that the $\mathcal{P}_{\mathsf{BATCH}}$ time (i.e., the time to aggregate b proofs) is $O(b \cdot \ell)$ and the $\mathcal{V}_{\mathsf{BATCH}}$ time (i.e., the time to verify the aggregated proof) is $O(b \cdot \ell)$. Unfortunately, even though our $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ argument uses the IPA with fast, $O(\log{(b \cdot \ell)})$-time KZG-based verification (see Section 3.3.4), the $\mathcal{V}_{\mathsf{BATCH}}$ verifier still needs to do $O(b \cdot \ell)$ work on the $\mathbf{B}_i$ vectors. (Note that this $O(b \cdot \ell)$ verifier work seems inherent for processing the b verification keys.) Lastly, the argument size (i.e., the aggregated proof size) is $O(\log{(b \cdot \ell)}) = O(\log b + \log \ell)$.

Cross-aggregation. In addition to aggregating proofs w.r.t. the some digest C, we can also *cross-aggregate* proofs w.r.t. different digests [72]. Specifically, suppose we have b proofs π_i for positions p_i, each w.r.t. a (potentially-different) digest C_i for a vector with MLE f_i. Then, we can use the same $\mathcal{P}_{\mathsf{BATCH}}$ prover from Fig. 3.5 to cross-aggregate these proofs. To verify, the verifier now computes the Z_i's given to $\mathcal{V}_{\mathsf{BATCH}}$ by using the right digest and evaluation point: i.e., $Z_i = e(C_i/g_1^{f_i(p_i)}, g_2)$.

3.4.4 Unstealable proofs

In this subsection, we show how to incentivize proof computation by allowing provers, who store the vector and maintain proofs, to *watermark* the proofs they compute. Such watermarked proofs are cryptographically-bound to their prover's identity, which means the prover can be monetarily rewarded for having computed them (e.g., in cryptocurrencies). Importantly, this cryptographic binding cannot be undone by adversaries. In other words, "stealing" a proof by replacing its watermark with your own, is no easier than computing the proof from scratch like everyone else. We call such watermarked proofs *unstealable*,

formalize and prove their security and make Hyperproofs unstealable. We show why and how unstealability is helpful in the cryptocurrency setting in Section 3.5. We also envision other applications could benefit from it.

Unstealability goals. First, any vector $\mathbf{a}$ should continue to have a single digest C against which all correct proofs verify, whether proofs are watermarked or not. Put differently, unstealability must work in our previous setting where there is a single Com algorithm for everyone, which does not take the identity of the prover as input. Specifically, only the Open and Ver algorithms are given the identity of the prover to watermark proofs and verify them. This ensures compatibility with stateless cryptocurrencies, where the state must have a single (prover-independent) digest against which (prover-dependent) watermarked proofs can be verified. Second, a prover should still be able to precompute all its (now) unstealable proofs and efficiently maintain them over time as the vector changes. In particular, solutions that require provers to watermark proofs "on the fly" would be too expensive. Third, unstealable proofs should remain aggregatable.

Strawmen. One idea for unstealability is to have each prover commit to the original vector $\mathbf{a}$ but "extended" with its identity id as $\mathbf{a}_{\mathsf{id}} = (a_i||\mathsf{id})_{i\in[n]}$. Unfortunately, this results in having multiple, prover-specific digests C_{id} for $\mathbf{a}$. Another idea is to add a digital signature on the VC proof. However, the signature can simply be removed by the adversary and replaced with their own. A last attempt would be to use a non-malleable SNARK [10] to augment a VC proof with a proof of knowledge of (1) the committed vector and (2) a secret associated with the prover's identity. This would require a stealing adversary to maul the

SNARK proof so as to verify for their identity. However, this approach would be too slow and would not preserve maintainability due to the non-malleability of the SNARK.

Unstealability via exponentiations. We make a proof $\boldsymbol{\pi}_i = (w_{i,\ell}, \ldots, w_{i,1})$ unstealable by exponentiating it with α as:

$$\boldsymbol{\pi}_i^\alpha = (w_{i,\ell}^\alpha, \ldots, w_{i,1}^\alpha) \stackrel{\mathsf{def}}{=} (\hat{w}_{i,\ell}, \ldots, \hat{w}_{i,1}), \tag{3.25}$$

where α is the prover's *watermarking secret key (WSK)*. The corresponding *watermarking public key (WPK)* is g_2^α together with a zero-knowledge proof of knowledge (ZKPoK) of α (e.g., a Schnorr proof [123] as per Section 3.8).

To verify a proof watermarked with g_2^α, one first checks that the ZKPoK of α verifies and that $\alpha \neq 0$. Second, one checks the proof as normal as per Eq. (3.20), but accounts for the WPK g_2^α:

$$e(\mathsf{C}/g_1^{a_i}, g_2^\alpha) \stackrel{?}{=} \prod_{k\in[\ell]} e(\hat{w}_{i,k}, g_2^{s_k - i_k}) \tag{3.26}$$

The ZKPoK of α is used to prevent stealing by exponentiating $\boldsymbol{\pi}_i^\alpha$ with a δ known by the adversary, since the adversary would have to prove knowledge of $\alpha \cdot \delta$. As a result, the adversary's only recourse is to remove α from the watermarked proof, but this seems to require exponentiating by α^{-1}, which the adversary does not know. We prove security in the *algebraic group model (AGM)* [65] in the extended version of our paper [127].

Aggregation-preserving unstealability. One important property of our unstealable proofs is that they remain aggregatable via a call to Agg from Fig. 3.6. Intuitively, this

is because the right-hand side of the watermarked verification from Eq. (3.26) remains the same as for normal verification in Eq. (3.20). However, the left-hand side changes. Thus, when verifying an aggregated proof via Ver in Fig. 3.6, the verifier has to account for the WPKs when computing the Z_i's given to $\mathcal{V}_{\mathsf{BATCH}}$ in Fig. 3.5. In other words, the verifier needs to have these WPKs. In our application setting from Section 3.5, we anticipate the verifier will already have all of the WPKs, instead of receiving them with the aggregated proof.

Homomorphism-preserving unstealability. Our approach to watermarking proofs preserves the PST and MLT homomorphisms. This has a few advantages. First, watermarked proofs can still be updated. Specifically, assuming position u changed by δ, the watermarked proof $\boldsymbol{\pi}_i^{\alpha}$ from Eq. (3.25) can be updated as before (see Eq. (3.18)) if one uses *watermarked update keys* $(\mathsf{upk}_{u,k})^{\alpha}$. Second, an MLT of watermarked proofs can be computed directly, if the prover uses *watermarked public parameters*. The prover can obtain these in a one-time pre-processing step that exponentiates all parameters from Fig. 3.1 with the WSK α:

$$\hat{g}_1^{\mathcal{S}_{u,k}(\mathbf{s})} \stackrel{\mathsf{def}}{=} \left(g_1^{\mathcal{S}_{u,k}(\mathbf{s})}\right)^{\alpha} = g_1^{\alpha\mathcal{S}_{u,k}(\mathbf{s})}, \forall k \in [0, \ell], \forall u \in [0, 2^k) \tag{3.27}$$

Importantly, these are still valid Hyperproofs parameters, except under a new base $\hat{g}_1 = g_1^{\alpha}$. As a result, the prover can directly compute a *watermarked MLT* using these new parameters. This is important, as it allows precomputing watermarked proofs, ensuring that serving such proofs is as efficient as serving normal proofs. Third, a watermarked MLT

is efficiently maintainable, just like a normal MLT. This follows from the updatability of watermarked proofs argued above.

New UVC algorithms. We must slightly change our VC API from Definition 3.3.1 into an *unstealable VC (UVC)* API that accounts for watermarked proofs and watermarking key-pairs. First, we introduce two new algorithms:

1. $\mathsf{WtrmkGen}(1^\lambda) \to (\mathsf{wsk}, \mathsf{wpk})$: Generates a random $(\mathsf{wsk}, \mathsf{wpk})$ watermarking key pair.
2. $\mathsf{WtrmkParams}(\mathsf{pp}, \mathsf{wsk}) \to \mathsf{wpp}$: Returns watermarked public parameters wpp, under wsk, as per Eq. (3.27).

Second, the algorithm $\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i, \mathsf{wpk}_i)_{i \in I}, \pi_I)$ additionally takes as input the watermarking PK wpk_i that each proof π_i is watermarked with. Third, the algorithms for creating and updating proofs now operate on *watermarked* public parameters. In the interest of brevity, we give the full UVC API, with a new correctness definition in Definition 3.9.3.

UVC soundness. We model UVC soundness similar to VC soundness, except we account for watermarked proofs. Informally, we prevent adversaries from creating two inconsistent proofs for the same position k, even if those proofs are watermarked with different, adversarially-generated WPKs (see the extended version [127]).

UVC unstealability. In Definition 3.9.6, we formalize our notion of unstealability which captures that an adversary cannot compute a watermarked proof on a WPK it knows asymptotically any faster than the Open algorithm, despite having adaptive access to a *watermarking oracle* on arbitrary choices of WPK. We prove that Hyperproofs is unstealable in the *algebraic group model (AGM)* [65].

In particular, we show that an adversary who outputs a new watermarked proof (after having access to the watermarking oracle) and runs asymptotically faster than the time it takes to run Open (thus breaking Definition 3.9.6) can neither explicitly include the oracle proofs in the output watermarked proof (or otherwise discrete log is broken) nor use the oracle proofs in any other way to speed up computation (or otherwise q-SDH is broken).

3.5 Hyperproofs for Cryptocurrencies

In this section, we discuss how Hyperproofs can be used to speed up validation in *payment-only* stateless cryptocurrencies.

Stateless validation. In account-based cryptocurrencies [156], *validators* such as miners and P2P nodes store a large amount of *state* to validate transactions and blocks in the consensus protocol. This state consists of each user's account balance and can be represented as a vector that maps each user's public key to their balance. Recent work [26, 57, 72, 98, 121, 138, 141] trades off storage of the state with additional bandwidth and computation. This approach, known as *stateless validation*, commits to the state using a vector commitment (VC) scheme and allows validators to store only the digest rather than the full state. Next, transactions and blocks are augmented with proofs for the accessed state, so validators can check validity against the digest, instead of storing the full state.

Practical relevance. We believe stateless validation addresses two important problems in cryptocurrencies. First, in smart-contract-based cryptocurrencies, every block validator in the network has to store the full state in order to validate. This leads to a *state explosion*

problem [37, 102], which could be ameliorated by having validators store succinct digests of the state. Then, each smart contract owner could store its own state and maintain its VC proofs, as proposed in previous work [72].

Second, in sharded cryptocurrencies, validators have to be frequently shuffled between shards to prevent adversaries from corrupting a majority of validators within a shard [87]. However, shuffling a validator from shard A to shard B requires that validator to download shard B's state. This worsens performance, but could be ameliorated by statelessly validating against a digest of the shard's state. As a result, when moving to shard B, a validator need only download that shard's digest, which is very small.

Challenges. There are several challenges in stateless validation. First, when creating a transaction, the sending user needs to include a proof that they have enough balance. In this sense, users should be able to fetch their proof from *proof-serving nodes (PSNs)* [121, 141], who maintain (a subset of) all proofs. Thus, *PSNs should be able to efficiently update all proofs*, as new blocks are confirmed. Second, *PSNs should be incentivized to maintain proofs*. Third, a miner must now include each transaction's proof in a proposed block, so that other miners can statelessly validate this block. This calls for *proofs to be efficiently aggregatable*, to save block space. Finally, when validating a block, miners must now verify such an aggregated proof. Thus, *aggregated proofs should verify fast*.

Why rely on proof-serving nodes? In theory, each user can maintain their proof locally by keeping up with all confirmed transactions and updating their proof (e.g., as per Eq. (3.18)). However, this overwhelms users with large computation (i.e., updating proofs) and large communication (i.e., downloading new blocks). This is why well-incentivized,

efficient proof-serving nodes (PSNs) are important: they eliminate this burden from users by allowing them to fetch their latest proof. We discuss below how unstealability helps implement well-incentivized PSNs.

Hyperproofs for stateless validation. As described above, in the stateless validation setting, it is important to minimize the time for (1) PSNs to update all proofs to reflect the latest block, (2) miners to propose a new block, with aggregated proofs and (3) validators (i.e., miners and P2P nodes) to verify this block, including its aggregated proof. In Section 3.6.3, we show experimentally that Hyperproofs outperforms other VCs in this task. This is because VCs with $O(1)$-sized proofs [42, 46, 72, 90, 141] require $O(n)$ time to update all proofs, while Hyperproofs only requires $O(\log n)$. Furthermore, when compared to Merkle trees, aggregation is 10× to 41× faster in Hyperproofs (see Section 3.6.2).

How unstealable proofs help. As highlighted above, proof-serving nodes should be rewarded for the proofs they serve. One approach would be for users to simply pay the PSN *before* they receive their proof. Unfortunately, this is vulnerable to a fair-exchange problem: a malicious PSN will take the payment but not send the proof. An alternative approach would be for PSNs to first serve the proof and expect payment *after*. This approach poses two challenges.

First, we must ensure the payment always goes through. Fortunately, this can be achieved via the cryptocurrency's consensus mechanism. Specifically, the miners can enforce a *PSN fee* whenever a valid PSN proof is included in a block. Second, and most importantly, we must ensure that the payment goes only to the PSN who served the proof. This requires that a proof served by PSN A cannot be mauled to appear as a proof served by (a malicious)

Table 3.2: Single-threaded microbenchmarks for Hyperproofs. Running times with an asterisk symbol (*) are too long and have been interpolated. We measure aggregation of 1024 proofs. OpenAll and Com are only measured once. UpdDig and UpdAllProofs times are averages after a batch of 1024 changes to the vector. All algorithms are parallelizable.

$\ell = \log_2 n$	**22**	**24**	**26**	**28**	**30**
Com (min)	3.1	12.6	50	201*	807*
OpenAll (hrs)	0.7	2.7	12*	52*	225*
UpdDig			47.76 μs		
UpdAllProofs (ms)	1.74	1.96	2.15	2.37	2.58
Indiv. Ver (ms)	8.15	8.22	9.10	10.09	10.93
Agg (s)	105	109	114	118	123
Aggr. Ver (s)	13	14	15	16	17
Indiv. proof size (KiB)	1.06	1.15	1.25	1.34	1.44
Aggr. proof size (KiB)			51.6		

PSN B. In other words, PSN B should have no recourse but to compute a proof from scratch like everyone else. Our unstealability design from Section 3.4.4 guarantees exactly this property.

3.6 Evaluation

In this section, we measure the performance of Hyperproofs and explore their applicability for stateless validation. We do not directly compare to VCs with constant-sized proofs due to their impractical $O(n)$ cost to update all proofs (see Section 3.6.1). Instead, we focus on Merkle trees with SNARK-based aggregation. We use the Golang bindings of the `mcl` library [99] to implement Hyperproofs[1]. We use BLS12-381, a pairing-friendly elliptic curve, which offers 128 bits of security. A serialized $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ element in `mcl` takes 48, 96, and 576 bytes, respectively. A single exponentiation takes 106 μs in $\mathbb{G}_1$ and 250 μs in

[1]Our code is available at: https://github.com/hyperproofs/hyperproofs

Table 3.3: The size of the public parameters from Fig. 3.1 for various values of $\ell = \log_2 n$. Recall that the *verification key* consists of all selector monomial commitments $g_2^{s_k}, \forall k \in [\ell]$, while the *proving key* consists of all selector multinomial commitents $g_1^{\mathcal{S}_{j,k}(\mathbf{s})}, \forall k \in [0, \ell], j \in [0, 2^k)$ (see Fig. 3.1).

$\ell = \log_2 n$	Verification key	Proving key
22	2.11 KiB	384 MiB
24	2.3 KiB	1.5 GiB
26	2.49 KiB	6 GiB
28	2.68 KiB	24 GiB
30	2.88 KiB	96 GiB

$\mathbb{G}_2$. Each experiment ran *single-threaded* on an Intel Core i7-4770 CPU @ 3.40GHz with 8 cores and 32 GiB of RAM. Unless stated otherwise, we perform 4 runs of each experiment and report their average. Also, vectors in this section are of size $n = 2^\ell$.

3.6.1 Microbenchmarks

We microbenchmark Hyperproofs in Table 3.2. All microbenchmarks pick vectors and updates randomly and are *single-threaded*, but trivially parallelizable.

Public parameters. To commit to vectors of size n, Hyperproofs needs a large *proving key* consisting of $2n-1$ $\mathbb{G}_1$ elements depicted in Fig. 3.1. For $\ell = 28$, this requires around 24 GiB of space (see Table 3.3). *Verification keys* are all derived from $(g_2^{s_k})_{k\in[\ell]}$. Furthermore, to aggregate b proofs, Abe et al. commitment keys [3] are needed consisting of $\ell \cdot b$ $\mathbb{G}_1$ and $\ell \cdot b$ $\mathbb{G}_2$ elements. For $\ell = 28$ and $b = 1024$, this only adds 3.94 MiB. Watermarking the public parameters as per Section 3.4.4 requires $2n - 1$ exponentiations in $\mathbb{G}_1$. For $\ell = 28$, this takes 15.87 hours. However, this is a one-time cost.

Committing and computing multilinear trees. We commit to a vector of size n via an $O(n)$-sized multi-exponentiation. For $\ell = 28$, this takes 202 minutes. Computing a multilinear tree (MLT) involves committing to the MLEs in each node via a multi-exponentiation (see Fig. 3.3). For $\ell = 28$, this takes 52.2 hours (or 1.63 hours with 32 threads). We expect to at least double performance via faster multi-exponentiation algorithms, which `mcl` lacks.

Updating the digest and the multilinear tree. For updating the digest, we measure the time to apply a batch of 1024 updates via a multi-exponentiation, divide this time by 1024 and obtain an average time of 48 μs per update. For the MLT, we also measure the time to apply a batch of 1024 updates. This way, we can use multi-exponentiations when updating nodes in the tree. Dividing the total time by 1024, gives us an average time of 1.74 ($\ell = 22$) to 2.58 milliseconds ($\ell = 30$) per update. Recall from Section 3.4.4 that updates will be just as fast for watermarked multilinear trees.

Proof size and verification time. Individual proof size is ℓ $\mathbb{G}_1$ elements and is competitive with Merkle trees (e.g., for $\ell = 30$, 1.44 KiB in MLTs versus 960 bytes in MHTs). Verifying a proof requires $\ell + 1$ pairings, which we optimize into a multi-pairing (i.e., first compute $\ell + 1$ Miller loops and then compute a single final exponentiation). This way, verifying a proof ranges from 8.2 ($\ell = 22$) to 11 milliseconds ($\ell = 30$). If the proof is watermarked, we discount the WPK from the proof size, since the verifier could already have the WPK, depending on the application. Furthermore, this overhead would be acceptable: 224 bytes (see Section 3.8). Lastly, verifying the ZKPoK for the WPK requires two $\mathbb{G}_2$ exponentiations, which adds around 500 μs to the proof verification time.

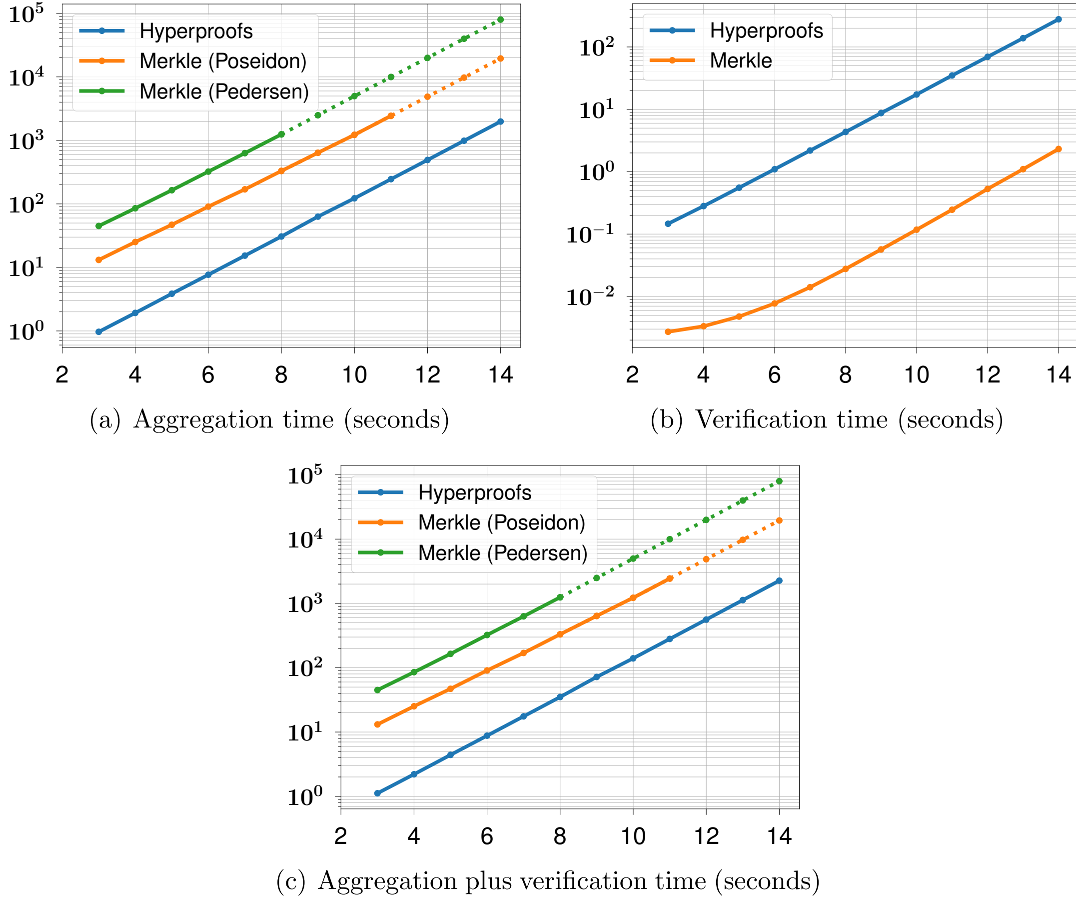
Proof aggregation. Let I be the set of transactions to be aggregated via $\mathsf{Agg}_{\mathsf{pp}}$, which calls $\mathcal{P}_{\mathsf{BATCH}}$ from Fig. 3.5. In our benchmarks, $b = |I| = 1024$. As shown in Table 3.2, aggregating 1024 transactions takes between 105 ($\ell = 22$) to 123 seconds ($\ell = 30$). Verifying such an aggregated proof takes between 13 ($\ell = 22$) to 17.5 ($\ell = 30$) seconds. These times are not affected by watermarking. In Section 3.6.2, we show our aggregation is 10× to 41× faster than SNARKs.

Aggregated proof size. Our aggregated proof size is 52 KiB for any $\ell = 22, \ldots, 30$. This is an artifact of the IPA proof size depending on the smallest power of two $\geq \log(b \cdot \ell)$, which is the same for all ℓ's above when $b = 1024$. As with individual proofs, watermarking does not affect proof size when the verifier has the WPKs.

Comparison with Pointproofs. One of the main advantages over aggregatable VCs with constant-sized proofs such as Pointproofs is that Hyperproofs are *maintainable*. For example, in Pointproofs, updating all $n = 2^\ell$ proofs involves n exponentiations, taking 31.7 hours for $\ell = 30$. Importantly, multi-exponentiations cannot be used here. In contrast, Hyperproofs only takes 3.2 milliseconds. (Unlike the numbers from Table 3.2, these numbers assume no batching.)

3.6.2 Comparison with SNARKs

In this subsection, we show that Hyperproof aggregation is anywhere from 10× to 41× faster than Merkle proof aggregation via SNARKs (see Fig. 3.7), depending on the choice of hash function. This comes at the cost of larger proofs (52 KiB versus 192 bytes) and slower verification. Nonetheless, the end-to-end time to aggregate-and-verify remains around 10×



(a) Aggregation time (seconds)

(b) Verification time (seconds)

(c) Aggregation plus verification time (seconds)

Figure 3.7: SNARK-based Merkle proof aggregation versus Hyperproof aggregation. The x-axis is $\log_2$(# of proofs being aggregated). Dotted lines are extrapolated, due to the SNARK prover running out of memory. We use the 128-bit secure variant of Poseidon.

to 41× faster in Hyperproofs. We find this design trade-off to be a good one for stateless cryptocurrencies where, although fast verification is important, aggregation times cannot be too high (see Section 3.6.3).

Experimental setting. We fix the height of both the Merkle tree and our MLT to $\ell = 30$, and measure performance when aggregating $b \in \{2^2, 2^4, \ldots, 2^{14}\}$ proofs. We compare to an implementation by Ozdemir et al. [105] in Rust [104] which uses the state-of-the-art SNARK by Groth [76] to prove *knowledge* of changes to a Merkle tree, updating it from digest d to digest d'. To benchmark proof aggregation, we notice that proof aggregation

would involve half of the work done by the Ozdemir et al. prover, and directly use their code. This is because proving knowledge of k changes involves first verifying k Merkle proofs for the original values "inside the SNARK" and then updating the Merkle root with the changes, which also involves k Merkle path verifications. For the SNARK verifier, we directly measure its work, which involves an $O(b)$-sized $\mathbb{G}_1$ multi-exponentiation and 3 pairings.

Choice of Merkle hash function. Choosing a "SNARK-friendly" hash function for the Merkle tree can significantly reduce the prover time. In this sense, we use the recently-proposed Poseidon-128 hash function [74], which only requires 316 R1CS constraints per invocation inside the SNARK, but lacks sufficient cryptanalysis. As a more secure choice, we also use the Pedersen hash function [160] used in Zcash [18], which is collision-resistant under the hardness of discrete log, but requires 2753 constraints per invocation [105].

Proving time. The SNARK prover time is dominated by several multi-exponentiations and Fast Fourier Transforms (FFTs) of size linear in the number of R1CS constraints. For example, aggregating $b = 2^{10}$ proofs in a Poseidon-hashed Merkle tree of height $\ell = 30$, involves 10 million constraints. As a result, SNARK aggregation is very slow, taking 1224 seconds. In contrast, when aggregating b Hyperproofs, also in a height ℓ MLT, our IPA-based prover from Fig. 3.5 only computes $O(b\ell)$ pairings and $O(b\ell)$ $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ exponentiations. This only takes 123 seconds. On average, as shown in Fig. 3.7(a), aggregating Hyperproofs is 10× faster than aggregating Merkle-Poseidon proofs and 41× faster than Merkle-Pedersen.

Prover memory. The SNARK prover also requires memory at least linear in the number of constraints. As a result, on our machine with 32 GiB of RAM, SNARK aggregation runs out of memory when aggregating $\geq 2^{11}$ proofs with Poseidon hashing (20 million constraints) or $\geq 2^8$ proofs with Pedersen (23 million constraints). Nonetheless, we extrapolate the proving times in Fig. 3.7. In contrast, our IPA-based aggregation from Fig. 3.5 has a much lower memory footprint and never runs out of memory.

Verification time. In general, verifying a SNARK proof requires 3 pairings and a $\mathbb{G}_1$ multi-exponentiation of size equal to the number of verifier-provided inputs [76]. In particular, when aggregating b Merkle proofs, this multi-exponentiation will be of size $2b+1$, since the verifier must input the digest and the b leaves $(i, v_i)_{i \in I}$ being verified. We implement verification in Golang using `mcl` [99] and report the times in Fig. 3.7(b). (We cannot use the Ozdemir et al. code, since the verifier only inputs two digests and checks *knowledge* of b changes to the Merkle tree.) When aggregating $b = 2^{10}$ proofs, it takes 0.11 seconds to verify a SNARK proof and 17.4 seconds to verify an aggregated Hyperproof. While verification is slower in Hyperproofs, when accounting for both the time to prove and verify in Fig. 3.7(c), Hyperproofs are faster.

SNARKs without trusted setup. Recent SNARKs [32, 125, 161] are *transparent* (i.e., do not need a trusted setup). Even better, these SNARKs often have faster provers than pairing-based SNARKs. However, compared to Hyperproofs, they are still too slow, have larger proof sizes and consume too much memory. For example, aggregating $b = 2^{14}$ Merkle proofs requires 2^{28} R1CS constraints if using Poseidon hashes. The prover time would be around 2.58 hours using Spartan [125, Figure 7] and 1.53 hours using Virgo [161]. This

is close to 5× and 3× slower than Hyperproofs, respectively. The proof size would be around 1.83 MiB using Spartan and 350 KiB using Virgo (estimated using the open-source code of [161]). This is around 36× and 7× bigger than Hyperproofs, respectively. The performance is even worse with Pedersen hashes. Moreover, these transparent SNARKs are not as memory-efficient as Hyperproofs: Virgo scales to 2^{24} constraints, similar to pairing-based SNARKs (i.e., fails aggregating when $b \geq 2^{11}$ proofs) while Spartan scales to 2^{26} constraints (i.e., fails for $b \geq 2^{13}$). Lastly, other transparent arguments (e.g., STARKs [17], Aurora [20], Hyrax [146], Ligero [9]) have similar drawbacks. We defer to [125, 161] for a detailed discussion on trade-offs.

3.6.3 Macrobenchmarks

Our *single-threaded* experiments measure the VC-induced overheads of statelessly reaching consensus on a new block, as discussed in Section 3.5. This consists of three measurements. First, the *block proposal* time (**P**) to verify individual proofs, aggregate them into a new block and update the digest. Second, the *block validation* time (**V**) to verify the aggregated proof and the updated digest in this new block, as it propagates through the P2P network. In particular, we assume the P2P network has diameter $h = 20$. Third, the *proof maintenance* time (**M**) for a proof-serving node (PSN) to update all proofs after applying the updates from this new block, so that the next proposed block can use these proofs.

We estimate the VC overhead as $\mathbf{P} + (h \cdot \mathbf{V}) + \mathbf{M}$ and summarize our results in Table 3.4. Note that we account for P2P nodes not forwarding a block before validating it by multiplying $\mathbf{V}$ by h. Overall, Hyperproofs' overhead is 10× smaller than Poseidon-

hashed Merkle trees and $41\times$ smaller than Pedersen-hashed. This is because Merkle-based stateless validation involves a slower, more complex SNARK prover (discussed below). Furthermore, Hyperproofs remain competitive in terms of proof maintenance cost ($\mathbf{M}$).

Setting: We assume MLTs and Merkle trees of height $\ell = 30$ and blocks of 1024 transactions. We do not compare to VCs with $O(1)$-sized proofs, due to their large proof maintenance cost (i.e., 2^ℓ $\mathbb{G}_1$ exponentiations, or 31.7 hours).

Limitations: Our macrobenchmarks do not account for all the subtleties that would arise in a full prototype, such as communication overheads, or miners needing to update the proofs in the current block they are working on due to another competing block. They also do not account for the overhead of signature verification, which is not affected by the chosen VC scheme. Instead, they focus on the three key operations whose overheads should be minimized: block proposal, block validation and proof maintenance. Lastly, while we show Hyperproofs are faster than other VCs for stateless validation, we do not claim they make the stateless setting practical.

Block transitions with Hyperproofs versus Merkle trees. In a stateless cryptocurrency, the ith block stores the digest d_i of all users' balances at that point in time. When block $i+1$ arrives, it must prove that its new digest d_{i+1} correctly reflects the updated balances, after applying its transactions. With Hyperproofs, the block includes an aggregated proof for the balance of each user sending money. This way, a validator can ensure that a block is spending valid coins and then can compute d_{i+1} from d_i via UpdDig, subtracting coins from each sending user's account and adding coins to each receiving user.

With SNARK-based Merkle trees, it is not possible to update the digest d_{i+1} given the old digest d_i, the SNARK aggregation proof, and the changes in balances: the Merkle proofs for all the changed leaves are also needed as auxiliary information. But including these Merkle proofs in the block would defeat the point of aggregating them via SNARKs! Therefore, the SNARK circuit must be extended to also verify the transition between d_i and d_{i+1}. Specifically, the circuit additionally proves that d_{i+1} is obtained by applying the changes in the block to d_i. A block of b transactions involves $2b$ changes to the Merkle tree, and each change requires two Merkle path verifications inside the circuit. Therefore, the circuit involves $4 \cdot b$ Merkle path verifications ($4\times$ more expensive than the aggregation circuit from Section 3.6.2).

Block overhead. As described above, stateless cryptocurrency blocks additionally store the digest of the state and an aggregated proof for all transactions. Both Merkle trees and Hyperproofs have similar digest sizes (i.e., 32 bytes versus 48 bytes). However, aggregated Hyperproofs are 52 KiB, whereas SNARK-aggregated Merkle proofs are only 192 bytes. Nonetheless, relative to the size of the full block, Hyperproof overhead is modest and only decreases with larger blocks. Furthermore, we foresee optimizing the IPA from Fig. 3.8 to reduce the proof size. Lastly, using unstealability to incentivize proof-serving nodes (which Merkle trees do not support) adds 224 bytes of WPK overhead for each PSN involved in the block. As an alternative, if the set of PSNs is fixed or grows slowly, then WPKs can be stored as part of the public parameters of the system.

Transaction overhead. Transactions propagating through the P2P network in a stateless cryptocurrency need to include proofs. With Hyperproofs, this only requires a 1.44

Table 3.4: Single-threaded, stateless cryptocurrency macrobenchmarks that measure the time to prepare a block for proposal (**P**), to validate a proposed block (**V**) and to update all proofs (**M**) after a new block is seen. A block propagates through a P2P network of diameter $h = 20$. Trees have height $\ell = 30$ and blocks have 1024 transactions. A Poseidon-128 hash takes 113 μs using the `go-iden3-crypto` library [71]. A Pedersen hash takes 37 μs using the `sapling-crypto` library [118].

Scheme	Hyperproofs	Merkle w/ Poseidon	Merkle w/ Pedersen
Block proposal (**P**)	2.23 min	81 min	332 min
Block validation (**V**)	17.5 sec	0.18 sec	0.18 sec
Proof maintenance (**M**)	5.14 sec	4.7 sec	1.54 sec
Total $(\mathbf{P} + (h \cdot \mathbf{V}) + \mathbf{M})$	8 min	81 min	332 min

KiB proof for the sender's balance. With Merkle trees, whether Poseidon- or Pedersen-hashed, this requires two 960 byte proofs, or 1.875 KiB, one for the sender and one for the receiver. This is because, to update the Merkle root, the miner also needs the receiver's Merkle proof as auxiliary information, whereas in Hyperproofs the digest can be updated homomorphically.

Block proposal. With Hyperproofs, a miner proposing a block with 1024 transactions has to (1) verify 1024 individual Hyperproofs, (2) aggregate these proofs, (3) and update the digest. With Merkle trees, this remains the same, except steps (2) and (3) are done in the SNARK prover. Table 3.4 shows block proposal is 36× (Poseidon) to 149× (Pedersen) faster in Hyperproofs than in SNARKs due to faster aggregation/digest updates.

Block validation. To validate an incoming block, a miner has to verify its aggregated proof and check its digest was computed correctly via UpdDig. In Table 3.4, we see that SNARKs are 97× faster to verify than an aggregated Hyperproof of $b = 1024$ proofs, which require $O(b\ell)$ $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ exponentiations to verify. While SNARK verification also incurs $O(b)$ cost, this only involves a fast $\mathbb{G}_1$ multi-exponentiation. Nonetheless, when

considering the time to propose *and* validate a block ($\mathbf{P} + h \cdot \mathbf{V}$), Hyperproofs remains 10× (Poseidon) to 41× (Pedersen) faster.

Proof maintenance. Recall that having updated proofs ready to be served is important in stateless cryptocurrencies, since users need to fetch and include their proofs when sending a transaction to a miner. Fortunately, a PSN can update all proofs in $O(\ell)$ time in both Hypeproofs and Merkle trees. Table 3.4 gives the concrete *batch* update time after 1024 transactions (or 2048 changes to the tree). Batch-updating Merkle trees is slightly faster than applying each update sequentially, because each node in the Merkle tree need only be updated once, by recomputing a hash (i.e., 113 μs for Poseidon and 37 μs for Pedersen). In contrast, when batch-updating MLTs, each node still needs to be updated several times to account for all the leaves that changed underneath it, as per Eq. (3.18). While we optimize this using a multi-exponentiation, MLTs will be slightly slower.

3.7 Discussion

Selective versus adaptive security for PST commitments. Papamanthou et al. prove security under ℓ-SDH (see Assumption 3.8.2), but only in a *selective* sense. Specifically, the (selective) adversary, whose goal is to equivocate about $f(\mathbf{i})$, must first decide on an $\mathbf{i}$ and reveal it to the challenger [107, Appendix C.1]. In contrast, we prove adaptive security for any point on the Boolean hypercube. Specifically, the (adaptive) adversary reveals nothing to the challenger about the point $\mathbf{i}$ it equivocates on and, in our security proof, the reduction simply "guesses" this $\mathbf{i}$ (see Theorem 3.9.1). One negative consequence of this guessing is a security loss of $\log_2 n$ bits, which we hope to address in future work.

Large public parameters. Hyperproofs for vectors of size n require $O(n)$-sized public parameters (see Section 3.3.2) which must be generated via a *trusted setup*. In practice, this setup would have to be implemented securely as a multi-party computation (MPC) protocol [19, 29, 30]. Recently, cryptocurrency projects have demonstrated the viability of this approach at the scale of $n \approx 2^{27}$ [28, 63, 147]. We hope to scale these techniques to $n \approx 2^{30}$ in future work. Alternatively, large public parameters can be avoided by splitting a large vector up into k chunks and committing to each chunk. This saves a factor of k in the size of the public parameters but leads to a k-sized digest. Importantly, one can still aggregate proofs in such a chunked vector since Hyperproofs support cross-aggregation. Lastly, Hyperproofs can be modified to work in a decentralized setting where the trapdoor $\mathbf{s} = (s_1, \ldots, s_\ell)$ is secret-shared among a set of servers, similar to recent work for bilinear accumulators [79]. This precludes the need to generate $O(n)$ public parameters and could be useful for applications in the permissioned setting.

Choice of public parameters. The most popular account-based cryptocurrency, Ethereum, currently has less than 185 million accounts. Thus, it would be sufficient to set $\ell = 28$ in Hyperproofs. Furthermore, once Ethereum's consensus layer will partition accounts over 64 shards [58], a smaller $\ell = 22$ would suffice. To determine the maximum number of aggregated proofs b, we need only consider the maximum number of transactions in a block. For example, to handle $2\times$ more than the average number of transactions in a Bitcoin block, setting $b = 4000$ is more than adequate.

Trade-offs of stateless blockchains. Christ et al. show that a stateless blockchain system must, asymptotically, either the validation state or the number of proofs that must

be updated is linear in the size of the vector, even if the values in a constant fraction of indices change [52]. However, the maintainability property our work allows a PSN to keep the proofs up-to-date efficiently, and the unstealability property incentivizes proof maintenance. Thus, the users can go offline and delegate the overhead of maintaining up-to-date proofs to an untrusted PSN.

3.8 Assumptions, definitions, and primitives

Our work builds upon the inner product argument (IPA) by Bünz et al. (see Section 3.3.4), which in turn relies on Abe et al. commitments to group elements [2], which are secure under the *Symmetric-eXternal Diffie-Hellman (SXDH)* assumption defined below.

Assumption 3.8.1 (SXDH)**.** *Let* $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda)$*. The Decisional Diffie-Hellman (DDH) problem in* $\mathbb{G}$ *is to decide whether* $c = ab$*, given* (g, g^a, g^b, g^c) *where* $g \in_R \mathbb{G}$*. The SXDH assumption is that DDH holds in both* $\mathbb{G}_1$ *and* $\mathbb{G}_2$*.*

We prove Hyperproofs satisfy soundness, as per Definition 3.9.2, under *q-Strong Diffie-Hellman (q-SDH)* assumption, defined below.

Assumption 3.8.2 (q-SDH [24])**.** *For any* PPT *adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c} (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda), s \in_R \mathbb{Z}_p^*, \\ \mathsf{pp} = ((p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2), g_2^s, g_1^s, \ldots, g_1^{s^q}) : \\ (a, g_1^{\frac{1}{s+a}}) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

The faster verifier for the Bünz et al. IPA from Section 3.3.4 relies on a modified Abe et al. commitment scheme which uses "structured" commitment keys. This modified com-

mitment scheme is binding under the *q-Auxiliary Structured Double Pairing (q-ASDBP)* assumption in $\mathbb{G}_1$ and $\mathbb{G}_2$ introduced in [34]. First, we present this assumption in $\mathbb{G}_2$ below.

Assumption 3.8.3 (q-ASDBP$_{\mathbb{G}_2}$)**.** *For any* PPT *adversary* $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^\lambda), \beta \in_R \mathbb{Z}_p^*, \\ \mathsf{pp} = ((p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2), g_1^\beta, \left(g_2^{\beta^{2i}}\right)_{i\in[1,q)} \\ (A_0, \ldots, A_{q-1}) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ (A_0, \ldots, A_{q-1}) \neq \mathbf{1}_{\mathbb{G}_1} \wedge 1_{\mathbb{G}_T} \neq \prod_{i=1}^{q-1} e(A_i, g_2^{\beta^{2i}}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Second, the $\mathbb{G}_1$ variant of q-ASDBP is defined similarly by swapping $\mathbb{G}_2$ with $\mathbb{G}_1$. Third, the q-ASDBP assumption holds in the generic group model [34].

Definition 3.8.1 (Non-interactive arguments of knowledge)**.** *Let $\mathcal{L}$ be an NP relation such that $\mathbf{x} \in \mathcal{L}$ if, and only if, there exists a witness $\mathbf{w}$ such that $\mathcal{L}(\mathbf{x}; \mathbf{w}) = 1$. A non-interactive argument of knowledge for $\mathcal{L}$ (e.g., SNARKs [111]) allows a verifier to efficiently verify that $\mathbf{x} \in \mathcal{L}$, without using $\mathbf{w}$, but via a (small) proof provided by an untrusted prover. A non-interactive argument of knowledge consists of three* PPT *algorithms, $(\mathcal{G}, \mathcal{P}, \mathcal{V})$:*

1. *$(PK, VK) \leftarrow \mathcal{G}(1^\lambda, \mathcal{L})$: Generates the proving and verification key for the program $\mathcal{L}$.*

2. *$\pi \leftarrow \mathcal{P}(PK, \mathbf{x}; \mathbf{w})$: Generates a proof π to prove that there exists $\mathbf{w}$ such that $\mathcal{L}(\mathbf{x}; \mathbf{w}) = 1$.*

3. *$\{0, 1\} \leftarrow \mathcal{V}(VK, \pi, \mathbf{x})$: Checks if the proof π is valid for $\mathbf{x}$ using the verification key VK.*

Definition 3.8.2 (Knowledge soundness)**.** *We say that an argument of knowledge* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *for NP relation* $\mathcal{L}$ *has* knowledge soundness *if, for any* PPT $\mathcal{A}$*, there is a* PPT *extractor* $\mathcal{E}$ *such that:*

$$\Pr\left[1 \leftarrow \mathcal{V}(VK, \pi, \mathbf{x}) \;\middle|\; \begin{array}{c} (PK, VK) \leftarrow \mathcal{G}(1^\lambda, \mathcal{R}), \\ (\pi, \mathbf{x}) \leftarrow \mathcal{A}(PK, VK), \\ \mathbf{w} \leftarrow \mathcal{E}(PK, VK, \pi, \mathbf{x}), \\ \mathcal{L}(\mathbf{x}; \mathbf{w}) \neq 1 \end{array} \right] \leq \mathsf{negl}(\lambda)$$

Observation: The intuition behind Definition 3.8.2 is that, if the verifier accepts a proof for $\mathbf{x}$, then the prover must "know" a witness $\mathbf{w}$ for $\mathbf{x}$ and therefore $\mathbf{x} \in \mathcal{L}$. Knowledge is modeled by an extractor $\mathcal{E}$ that can output such a $\mathbf{w}$ by inspecting the prover's tape.

Zero-knowledge proofs of knowledge (ZKPoKs). Recall from Section 3.4.4 that a watermarking public key (WPK) g_2^α must come with a ZKPoK of α. For this, we rely on Schnorr ZKPoKs of α [123], defined as $\mathsf{zkpok}_\alpha = (z, c) \in \mathbb{Z}_p^2$, where:

$$\kappa \in_R \mathbb{Z}_p, \quad y = g_2^\kappa, \quad c = H(g_2, g_2^\alpha, y), \quad z = \kappa + c \cdot \alpha$$

To verify zkpok_α, one checks if:

$$\begin{aligned} c &\stackrel{?}{=} H(g_2, g_2^\alpha, g_2^z/(g_2^\alpha)^c) \stackrel{?}{=} H(g_2, g_2^\alpha, g_2^{(\kappa + c\cdot\alpha)}/g_2^{c\cdot\alpha}) \Leftrightarrow \\ &\stackrel{?}{=} H(g_2, g_2^\alpha, g_2^\kappa) = H(g_2, g_2^\alpha, y) = c \end{aligned} \tag{3.28}$$

where $H(\cdot)$ is modeled as a random oracle.

Inner Product Arguments (IPA). We give the *interactive* variant of the Bünz et al. [34] IPA in Fig. 3.8. Here, the prover interacts with the verifier over $\log m$ rounds. This interactive IPA is knowledge-sound assuming Abe et al. commitments are binding [34, Theorem 1]. It is made non-interactive via the Fiat-Shamir transform [62] and proved secure in a new *algebraic commitment model* and in the *random oracle model (ROM)* [34, Appendix D.2].

Faster verifier. As described Fig. 3.8, the prover and verifier take $O(m)$ time and the proof size is $|\pi| = O(\log m)$, if the argument is made non-interactive via the Fiat-Shamir transform [62]. However, Bünz et al. show how to reduce the verifier time by using a "structured" commitment key $\mathbf{ck} = (\mathbf{v}, \mathbf{w})$, similar to a q-SDH common reference string (see Assumption 3.8.2). This allows the verifier to outsource the computations of $\mathbf{v}'$ and $\mathbf{w}'$ to the untrusted prover and reduces verification time from $O(m)$ to $O(\log m)$. However, this comes at the cost of additionally relying on the q-SDH assumption (see Assumption 3.8.2) and the q-ASDBP assumption (see Assumption 3.8.3). Our work implicitly assumes this optimized verifier, which we later implement in Section 3.6. We refer the reader to [34, Section 5] for the details of this optimization, which is beyond the scope of this paper.

Lemma 3.8.1 (Random linear combinations lemma)**.** *Let $Z_i \in \mathbb{G}_T$, $\mathbf{A}_i \in \mathbb{G}_1^m$ and $\mathbf{B}_i \in \mathbb{G}_2^m$ for $i = 1, \ldots, N$. Assume each r_i is chosen uniformly at random from $\mathbb{Z}_p$. Then, with probability at least $1 - 1/p$, all Eq. (3.23) are satisfied iff. Eq. (3.24) is satisfied.*

Proof (sketch) for Lemma 3.8.1. Clearly if Eq. (3.23) are satisfied then Eq. (3.24) is also satisfied. For the other direction, by the Schwartz-Zippel lemma [124, 166], if at least one

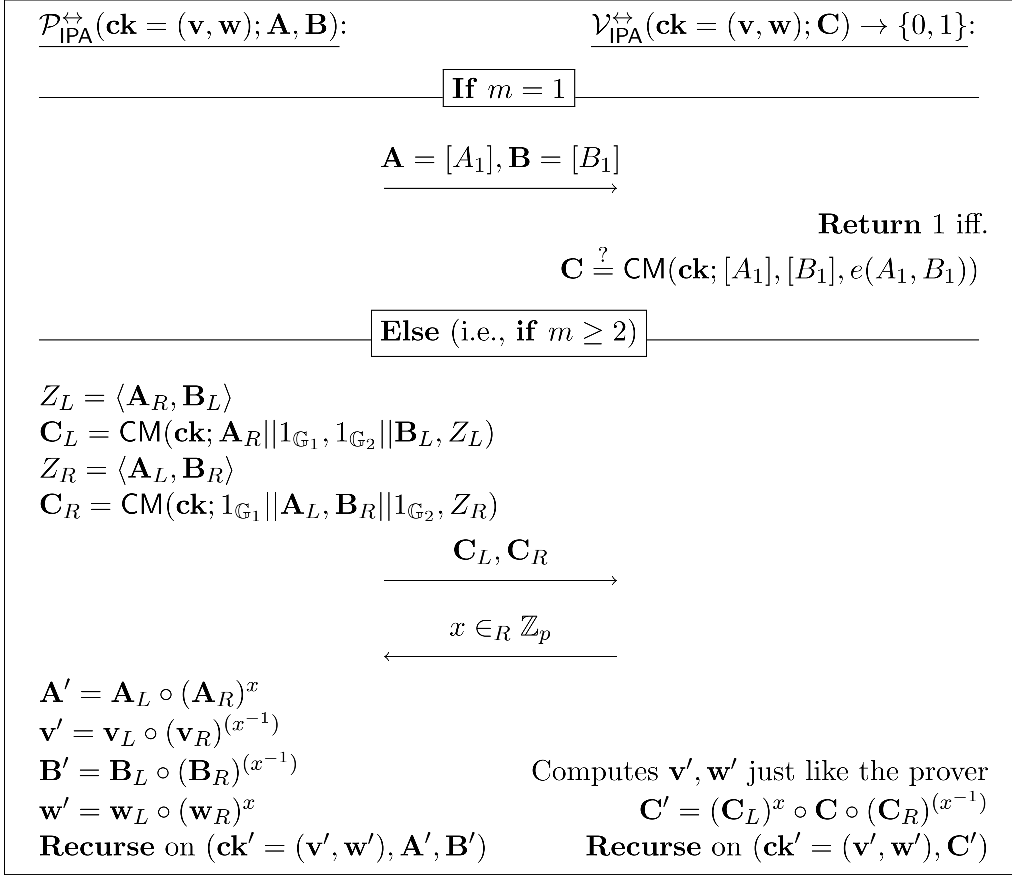


Figure 3.8: The *interactive* IPA by Bünz et al. for $m = 2^k$ (wlog). The prover convinces the verifier that he knows $(\mathbf{A}, \mathbf{B}) \in \mathbb{G}_1^m \times \mathbb{G}_2^m$ such that $\mathbf{A}, \mathbf{B}, Z$ are committed in $\mathbf{C}$ (under commitment key $\mathbf{ck}$) and that $Z = \langle \mathbf{A}, \mathbf{B} \rangle$. See Section 3.3 for IPA-specific notation such as $1_{\mathbb{G}_b}, \mathbf{A}_L, \mathbf{A} \circ \mathbf{B}, \mathsf{CM}, \mathbf{A}_R || 1_{\mathbb{G}}$ or $\mathbf{A}_L^x$.

equation from Eq. (3.23) does not hold, Eq. (3.24) holds at randomly selected values r_i with probability $1/p$, completing the proof. □

Theorem 3.8.1. $(\mathcal{G}_{\mathsf{BATCH}}, \mathcal{P}_{\mathsf{BATCH}}, \mathcal{V}_{\mathsf{BATCH}})$ *from Fig. 3.5 is a non-interactive argument of knowledge for* $\mathcal{L}^{b,\ell}_{\mathsf{BATCH}}$ *from Eq. (3.22) that has knowledge soundness according to Definition 3.8.2 under the same assumptions as the non-interactive IPA from Section 3.3.4 (i.e., algebraic commitment model [34], the random oracle model,* $(2b\ell)$*-SDH,* $(b\ell)$*-ASDBP).*

Proof (sketch) for Theorem 3.8.1. This follows from Lemma 3.8.1 and the knowledge soundness of the Bünz et al. IPA, which is used in a black box fashion. □

3.9 VCs and Hyperproofs

MLT correctness. We argue below why our multilinear tree from Section 3.4.1 yields correct proofs as per Definition 3.9.1. Formally, the quotients $q_j(x_j, \ldots, x_1)$, $\forall j \in [\ell]$ for a proof π_i in our MLT from Fig. 3.3 are computed as:

$$q_j(x_{j-1}, \ldots, x_1) = f_{i_\ell,\ldots,i_{j+1},1} - f_{i_\ell,\ldots,i_{j+1},0} \tag{3.29}$$

One can prove that that these quotients satisfy the PST decomposition from Eq. (3.9), and thus yield a correct proof, for any i via induction. Here, we just show this intuitively. Begin with the first term in the PST decomposition sum from Eq. (3.9), which is $q_\ell(\mathbf{x})(x_\ell - i_\ell)$. By Eq. (3.29), this term is equal to:

$$\begin{aligned}
q_\ell(\mathbf{x})(x_\ell - i_\ell) &= (f_1(\mathbf{x}) - f_0(\mathbf{x}))\,(x_\ell - i_\ell) \\
&= x_\ell f_1(\mathbf{x}) - i_\ell f_1(\mathbf{x}) - x_\ell f_0(\mathbf{x}) + i_\ell f_0(\mathbf{x}) \\
&= (1 - x_\ell) f_0(\mathbf{x}) + x_\ell f_1(\mathbf{x}) - (1 - i_\ell) f_0(\mathbf{x}) - i_\ell f_1(\mathbf{x}) \\
&= f(\mathbf{x}) - f_{i_\ell}(\mathbf{x})
\end{aligned}$$

The next term in the sum from Eq. (3.9) would be $q_{\ell-1}(\mathbf{x})(x_{\ell-1} - i_{\ell-1})$ which, by similar reasoning, equals $f_{i_\ell}(\mathbf{x}) - f_{i_\ell,i_{\ell-1}}(\mathbf{x})$. Adding these two terms up, the $f_{i_\ell}(\mathbf{x})$'s cancel out, leaving us with $f(\mathbf{x}) - f_{i_\ell,i_{\ell-1}}(\mathbf{x})$. Continuing with the other terms, everything cancels out except for $f(\mathbf{x}) - f_{i_\ell,\ldots,i_1}(\mathbf{x}) = f(\mathbf{x}) - f(\mathbf{i})$, as per Eq. (3.9). Therefore, the quotients defined in our MLT from Fig. 3.3 are correct.

Definition 3.9.1 (VC Correctness). *A VC is correct, if for all* $\lambda \in \mathbb{N}$ *and* $n = \mathsf{poly}(\lambda)$, *for all* $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n)$, *for all vectors* $\mathbf{a} = [a_0, \ldots, a_{n-1}]$, *if* $\mathsf{C} = \mathsf{Com}_{\mathsf{pp}}(\mathbf{a})$ *and* $\pi_i = \mathsf{Open}_{\mathsf{pp}}(i, \mathbf{a}), \forall i \in [0, n)$ *(or from* $\mathsf{OpenAll}_{\mathsf{pp}}(\mathbf{a})$*), then, for any polynomial number of updates* (u, δ) *resulting in a new vector* $\mathbf{a}'$, *if* C' *and* π_i', *for all* i, *are the updated digest and proofs obtained via calls to* $\mathsf{UpdDig}_{\mathsf{pp}}$ *and* $\mathsf{UpdProof}_{\mathsf{pp}}$ *(or to* $\mathsf{UpdAllProofs}_{\mathsf{pp}}$*) respectively, then (1)* $\Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', \{i\}, a_i', \pi_i')] = 1$ *for all* i*; (2)* $\forall I \subseteq [n], \Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', I, (a_i')_{i \in I}, \mathsf{Agg}_{\mathsf{pp}}(I, (a_i', \pi_i')_{i \in I}))] = 1$.

Observation: At a high-level, correctness says that proofs created via Open or OpenAll verify successfully via Ver, even in the presence of updates and aggregated proofs.

Definition 3.9.2 (VC Soundness). $\forall$ PPT *adversaries* $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n), \\ (\mathsf{C}, I, J, (a_i)_{i \in I}, (a_j')_{j \in J}, \pi_I, \pi_J') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i)_{i \in I}, \pi_I) \wedge \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, J, (a_j')_{j \in J}, \pi_J') \wedge \\ \exists k \in I \cap J \text{ s.t. } a_k \neq a_k' \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: Soundness says that no adversary can output two *inconsistent proofs* for different values $a_k \neq a_k'$ at position k with respect to an adversarially-produced digest d. Note that such a definition allows the digest C to be produced adversarially. This is stronger than what is required in our cryptocurrency setting from Section 3.5, where the digest is produced correctly from the agreed transactions. Nonetheless, having a stronger definition makes our VC from Section 3.4 more widely useful.

Theorem 3.9.1 (Individual Hyperproofs are sound)**.** *Our* individual $\log n$*-sized (non-aggregated) proofs from Section 3.4.1 are sound as per Definition 3.9.2 under q-SDH (see Assumption 3.8.2).*

Proof. See the extended version of our paper [127] □

Theorem 3.9.2 (Aggregated Hyperproofs are sound)**.** *Our* aggregated *proofs from Section 3.4.3 are sound as per Definition 3.9.2 under the knowledge-soundness of the* $\mathcal{L}_{\mathsf{BATCH}}$ *argument (see Theorem 3.8.1).*

Proof. See the extended version of our paper [127] □

Definition 3.9.3 (UVC)**.** *An* unstealable VC (UVC) *scheme consists of the following algorithms:*

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{pp}$: Given security parameter λ and vector length n, it outputs public parameters pp.

$\mathsf{WtrmkGen}(1^\lambda) \to (\mathsf{wsk}, \mathsf{wpk})$: Outputs a randomly-generated watermarking secret key (WSK) wsk and its corresponding watermarking public key (WPK) wpk.

$\mathsf{WtrmkParams}(\mathsf{pp}, \mathsf{wsk}) \to \mathsf{wpp}$: Returns watermarked public parameters that can be used to directly compute watermarked proofs under wsk.

$\mathsf{Com}_{\mathsf{pp}}(\mathbf{a}) \to \mathsf{C}$: Outputs the digest C of $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_p^n$.

$\mathsf{Open}_{\mathsf{wpp}}(i, \mathbf{a}) \to \pi_i$: Outputs a proof π_i for position i in $\mathbf{a}$, watermarked with the WSK from wpp.

$\mathsf{OpenAll}_{\mathsf{wpp}}(\boldsymbol{a}) \to (\pi_0, \ldots, \pi_{n-1})$: Outputs all proofs π_i for $\mathbf{a}$, watermarked with the WSK from wpp.

$\mathsf{Agg}_{\mathsf{pp}}(I, (a_i, \pi_i)_{i \in I}) \to \pi_I$: Combines individual proofs π_i for values a_i into an aggregated proof π_I.

$\mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i, \mathsf{wpk}_i)_{i \in I}, \pi_I) \to \{0, 1\}$: Verifies proof π_I that each position $i \in I$ has value a_i against digest C. Additionally checks that the proof for a_i was watermarked using wpk_i.

$\underline{\mathsf{UpdDig}_{\mathsf{pp}}(u, \delta, \mathsf{C}) \to \mathsf{C}'}$: Updates digest C to C' to reflect position u changing by $\delta \in \mathbb{Z}_p$.

$\underline{\mathsf{UpdAllProofs}_{\mathsf{wpp}}(u, \delta, \pi_0, \ldots, \pi_{n-1}) \to (\pi'_0, \ldots, \pi'_{n-1})}$: Updates all watermarked proofs π_i to π'_i after changing position u by δ.

$\underline{\mathsf{UpdProof}_{\mathsf{wpp}}(u, \delta, \pi_i) \to \pi'_i}$: Updates watermarked proof π_i to π'_i after changing position u by δ.

3.9.1 Unstealable VCs

In Section 3.4.4, we briefly explained how to change our VC API from Definition 3.3.1 to account for unstealability. We give the full API for an UVC in Definition 3.9.3 and we give definitions of correctness, security and unstealability below. Changes from Section 3.3.3 are highlighted in blue.

Definition 3.9.4 (UVC Correctness)**.** *An unstealable VC (UVC) is correct, if for all* $\lambda \in \mathbb{N}$ *and* $n = \mathsf{poly}(\lambda)$, *for all* $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n)$, *for all vectors* $\mathbf{a} = [a_0, \ldots, a_{n-1}]$, *if* $\mathsf{C} = \mathsf{Com}_{\mathsf{pp}}(\mathbf{a})$ *and* $\pi_i = \mathsf{Open}_{\mathsf{wpp}_i}(i, \mathbf{a}), \forall i \in [0, n)$, *where* $\mathsf{wpp}_i = \mathsf{WtrmkParams}(\mathsf{pp}, \mathsf{wsk}_i)$ *and* $(\mathsf{wsk}_i, \mathsf{wpk}_i) = \mathsf{WtrmkGen}(1^\lambda)$ *(or from any combination of* $\mathsf{OpenAll}_{\mathsf{wpp}_i}(\mathbf{a})$ *calls, with different* wpp_i*'s), then, for any polynomial number of updates* (u, δ) *outputting a new vector* $\mathbf{a}'$, *if* C' *and* π'_i, *for all* i, *are the updated digest and proofs via calls to* $\mathsf{UpdDig}_{\mathsf{pp}}$ *and* $\mathsf{UpdProof}_{\mathsf{wpp}_i}$ *(or* $\mathsf{UpdAllProofs}_{\mathsf{wpp}_i}$*) respectively, then:*

1. $\Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', \{i\}, (a'_i, \mathsf{wpk}_i), \pi'_i)] = 1$ *for all* i
2. $\Pr[1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}', I, (a'_i, \mathsf{wpk}_i)_{i \in I}, \mathsf{Agg}_{\mathsf{pp}}(I, (a'_i, \pi'_i)_{i \in I}))] = 1$, $\forall I \subseteq [n]$.

Definition 3.9.5 (UVC Soundness)**.** *∀ PPT adversaries $\mathcal{A}$:*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, n), \\ \left(\mathsf{C}, \begin{array}{c} I, \pi_I, (a_i, \mathsf{wpk}_i)_{i\in I}, \\ J, \pi'_J, (a'_j, \mathsf{wpk}_j)_{j\in J} \end{array}\right) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, I, (a_i, \mathsf{wpk}_i)_{i\in I}, \pi_I) \wedge \\ 1 \leftarrow \mathsf{Ver}_{\mathsf{pp}}(\mathsf{C}, J, (a'_j, \mathsf{wpk}_j)_{j\in J}, \pi'_J) \wedge \\ \exists k \in I \cap J \text{ s.t. } a_k \neq a'_k \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Theorem 3.9.3. *The unstealable variant of Hyperproofs from Section 3.4.4 is sound as per Definition 3.9.5.*

Proof. See the extended version of our paper [127] □

Definition 3.9.6 (UVC Unstealability)**.** *Let the worst-case complexity of* Open *be $O(g(n))$. An UVC is unstealable if ∀ security parameters λ, $\forall n = \mathsf{poly}(\lambda)$, ∀ PPT adversaries $\mathcal{A}$ running in $o(g(n))$ time:*

$$\Pr\left[\ \mathsf{Unstealability}^{\mathcal{A}}_{\mathsf{UVC}}(\lambda, n) = 1\ \right] \leq \mathsf{negl}(\lambda)\,.$$

$\underline{\mathsf{Unstealability}^{\mathcal{A}}_{\mathsf{UVC}}(\lambda, n)}$:

1: $\mathbf{PP} \leftarrow \mathsf{Gen}(1^\lambda, n)$

2: $\mathbf{a} = [a_0, \ldots, a_{n-1}] \in_R \mathbb{Z}_p^n$

3: $\mathsf{C} \leftarrow \mathsf{Com}_{\mathbf{PP}}(\mathbf{a})$

4: Initialize oracle $\mathsf{O}^{\mathsf{watrm}}_{\mathbf{PP},\mathbf{a}}(\cdot,\cdot)$ with $\mathsf{S} = \varnothing$

5: $(i_F, \mathbf{WPK}^*, \mathbf{Q}^*) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{watrm}}_{\mathbf{PP},\mathbf{a}}(\cdot,\cdot)}(\mathbf{PP}, \mathbf{a})$

6: **return true** if and only if:

  1: $\mathsf{Ver}_{\mathbf{PP}}(\mathsf{C}, i_F, (a_{i_F}, \mathbf{WPK}^*), \mathbf{Q}^*) = 1 \wedge$

  2: $(\cdot, \mathbf{WPK}^*) \notin \mathsf{S}$

$\underline{\mathsf{O}^{\mathsf{watrm}}_{\mathbf{PP},\mathbf{a}}(o, \mathbf{WPK})}$:

1. **if** $\mathbf{WPK} = \perp$ **or** cannot find $(\mathsf{wsk}, \mathbf{WPK}) \in \mathsf{S}$
2. $\quad(\mathsf{wsk}, \mathbf{WPK}) \leftarrow \mathsf{WtrmkGen}_{\mathbf{PP}}(1^\lambda)$
3. $\quad\mathsf{S} \leftarrow \mathsf{S} \cup \{(\mathsf{wsk}, \mathbf{WPK})\}$
4. $\mathsf{wpp} \leftarrow \mathsf{WtrmkParams}(\mathbf{PP}, \mathsf{wsk})$
5. $\mathbf{Q} \leftarrow \mathsf{Open}_{\mathsf{wpp}}(o, \mathbf{a})$
6. **return** $(\mathbf{Q}, \mathbf{WPK})$

Theorem 3.9.4 (Hyperproofs Unstealability)**.** *Hyperproofs is unstealable as per Definition 3.9.6 in the AGM, under discrete log and q-SDH assumption.*

Proof. See the extended version of our paper [127] ∎

Chapter 4: Batching, Aggregation, and Zero-Knowledge Proofs in Bilinear Accumulators

An *accumulator* is an authenticated data structure for a set of elements. It allows a *prover* to provide a succinct binding digest to a set of elements and to generate a short proof of membership or non-membership for any element in the accumulator domain. A *verifier* can efficiently check the proof of (non-)membership using the digest without requiring access to the entire set. Accumulators have found numerous applications including timestamping [21], fail-stop signature schemes [15], anonymous credentials [6, 11, 40, 41], cloud storage [142, 164] and more recently, stateless and privacy-preserving cryptocurrencies [26, 50, 97].

Batching and aggregation. In traditional applications, accumulators have been used in a setting where the prover had to provide (non-)membership proofs for a single element at a time. However, in emerging applications, such as cryptocurrencies, a prover must simultaneously prove (non-)membership of multiple elements. Naively, the prover could include individual proofs for each element, but this imposes high bandwidth usage and computational cost on the verifier. A better approach is *a batch proof*, that is, a succinct proof for multiple elements, which can be used to efficiently and simultaneously prove (non-)membership of multiple elements. For example, in UTXO-based stateless blockchains [26,

50], all transactions are accompanied by a proof of membership in the UTXO set. If a block proposer naively includes all individual proofs for validation (instead of a single batch proof), the size of the blocks transmitted across the network increases along with the computational overhead on the verifiers.

Let X be the set of elements in an accumulator. A batch membership proof for a set of elements $I \subseteq X$ can be computed: (1) by using the *trapdoor* (e.g., factors of the modulus in the RSA setting), or (2) *from scratch* using *all* the elements in X, or (3) by *aggregating* previously computed individual proofs of elements in I. Unfortunately, computing the batch proof using the trapdoor is impractical as it requires a trusted accumulator manager to hold the trapdoor. Furthermore, if the trapdoor is compromised, an adversary can forge proofs at-will. Computing the batch proof using the entire set is also impractical in a *distributed setting*, as nodes may not have access to the entire set or the trapdoor or a trusted accumulator manager. Moreover, updates to the accumulator arrive in batches (e.g., batch of transactions in a block). In comparison with other approaches, computing the batch proof by *aggregating* individual proofs is useful and relevant in the distributed setting as it does not require the trapdoor or the entire set. This brings us to the first question: *Is it possible to efficiently aggregate individual (non-)membership proofs?*

Zero-knowledge proofs. Proofs of (non-)membership for accumulators are often required in applications where privacy is critical. As an example, consider the application of anonymous credentials for authentication where valid (i.e., non-revoked) credentials are stored in an accumulator. When users wish to prove something about their credential embedded attributes, they also need to prove that their credential is valid via a proof of membership.

Such proofs are usually done in a zero-knowledge (ZK) fashion in order to guarantee unlinkability between proving sessions and specific user credentials. More concretely, let x be a user credential accumulated in A. The user will compute a commitment $c = Commit(x)$, and prove, in zero-knowledge, membership of the committed value x. However, being able to simultaneously prove (non-)membership of a set of elements, I, is an important property in scenarios where a user/organization controls multiple credentials. Additionally, a prover may also need to argue that this set I is of at least some size d while hiding I. For instance an organization holding $|I| \geq d$ valid credentials might want to prove in ZK that it can cast up to d votes. Such questions become even more relevant in recently developed decentralized identity systems [1, 84]. This brings us to the second question: *Is it possible to efficiently prove knowledge of a set I that is a subset of/disjoint from X, without revealing I? And reveal a lower bound of $|I|$, if needed?*

Batching, aggregation, and ZK in RSA accumulators. One of the most popular accumulator instantiations is the *RSA accumulator* [21, 41, 92]. Given a set X of prime numbers $(x_1, \ldots, x_n)$, one can define the accumulator A_X as the RSA group element $g^{\prod x_i}$, and a membership proof w of $y \in X$ as the accumulator of $X \setminus \{y\}$, which can be verified by checking whether w^y equals A_X. Boneh et al. [26] defined batch proofs for the RSA accumulator and also provided aggregation algorithms for both membership and non-membership RSA proofs. The resulting batch proofs are non-interactive and of constant size. They also present a Proof-of-Exponentiation protocol (PoE) to concretely speed up batch verification by reducing the number of group operations from $O(|I|)$ to a constant. Without PoE, the verifier would need to perform a large exponentiation that grows with

the number of elements in the proof. On the ZK front, Ozdemir et al., introduced *improved* SNARK-friendly techniques to batch prove (non-)membership in the RSA setting [105]. Subsequently, Campanelli et al. adopted a "hybrid" approach where they prove the batch membership without SNARKs and prove that elements of the batch are from the prime domain using SNARKs [43]. However, their accumulator does not support both membership and non-membership simultaneously.

While RSA accumulators have been used in many applications, they do present some crucial limitations:

- First, RSA group elements are large and this affects verification time and proof sizes.
- Second, they only support accumulation of elements that reside in a prime domain. Thus, they cannot be directly used for the accumulation of arbitrary elements. [1]

These problems are fundamental and limit the use of RSA accumulators in the batch setting regardless of the privacy concerns.

The case for bilinear accumulators. As opposed to RSA accumulators, *bilinear-pairing accumulators* (BP) [103, 164] have much smaller proofs and faster exponentiations. Moreover, BP accumulators can support the accumulation of arbitrary elements, making them directly applicable to a broader set of applications. Given a set X of arbitrary numbers $(x_1, \ldots, x_n)$, the accumulator A is: $g^{\prod(s+x_i)}$, where g is a prime order group generator and s is a secret *trapdoor*. The membership proof w of $y \in X$ is the accumulator of $X \setminus \{y\}$, which can be verified by using the A_X and the pairing operator. As highlighted above,

[1] While there exist techniques to map arbitrary elements to primes (i.e., hash to primes [26]), such mappings can harm the soundness of the accumulator if not carefully implemented during verification, especially if the mappings are not 1-1.

we are interested in accumulators for the distributed setting, i.e., where nobody can have access to the secret trapdoor s or the accumulated set X. Existing batching approaches in bilinear setting are only secure under a weak soundness definition [45, 69, 134] (with [134] additionally missing some security arguments), or require large public parameters [164]. Moreover, to the best of our knowledge, no prior work allows for efficient aggregation of non-membership proofs or efficient zero-knowledge (non-)membership proofs in the batch setting for *trapdoorless* accumulators (no accumulator operation except the setup requires the trapdoor, and the trapdoor is destroyed after setup if used). This is the main focus of our work.

Contributions. In the distributed setting, we make the following contributions to trapdoorless BP accumulators:

1. We propose a new algorithm to aggregate individual non-membership proofs into a single constant sized proof (Section 4.4) and provide a constant-time algorithm to update the extended Euclidean based individual non-membership proofs (Section 4.5).

2. We implement the first efficient ZK scheme that proves batch (non-)membership of BP accumulator using the knowledge-of-exponent assumption (Section 4.7), which can additionally reveal a lower bound on the size of the batch witness without revealing the elements of the batch proof. Asymptotically, the batch zerk-knowledge proof size is constant, the verification cost is constant, and the prover is $O(d)$, where d is the size of the batched subset.

3. We perform an experimental evaluation and comparison of the RSA and BP accumulators in both ZK and non-ZK setting (Section 4.8). Concretely, we observe that in the ZK setting the prover is around $16\times$ faster than RSA based baseline.

   In the non-ZK setting, we benchmark the batching and aggregation of accumulator proofs. For the first time, we explicitly take the mapping cost (from arbitrary domain to accumulator domain) into account. We observe that membership and non-membership proofs in BP accumulators are $2.5\times$ to $5\times$ smaller and $3.5\times$ smaller than the RSA accumulators, respectively. Moreover, verification of aggregated BP accumulator proofs is on an average $4\times$ faster than the RSA accumulators.

4. Finally, in Section 4.6, we propose a PoE protocol in Elliptic Curve (EC) groups that help us speed up batch verification. Our PoE can prove exponentiation of an arbitrary group element and it concretely saves one exponentiation for the prover when compared to concurrent works [54,134]. We use our PoE to verify exponentiation of a group element by ℓ-degree polynomial without having to perform $O(\ell)$ group exponentiations, only $O(\ell)$ operations in the field.

We remark that the definition of soundness considered in this work is *stronger* than those considered in prior works, since we *do not* assume that the accumulator is well-formed and we allow the adversary to pick the accumulator value (Definition 4.2.2). This strong soundness definition is useful for many modern applications of accumulators, especially in the distributed/blockchain setting, as it is not always realistic to assume that the accumulator value is well-formed.

Implications of our results and evaluations. As indicated above, batch *membership* proofs in BP accumulators, clearly outperform their RSA counterparts in the size of the proofs (2.5× to 5×). This makes our protocols very appealing to applications where communication cost is critical.

An example of such a potential application is the extension of ℓ-bit Byzantine agreement (BA) and broadcast protocols (BB) [101] where bilinear-accumulators are used to obtain state-of-the-art communication costs for ℓ-bit BA/BB. Our batching techniques can improve their so called: "distribute phase" by batching the proofs sent between the participants which would result in a constant size proof and significantly reduce their communication costs.

In the ZK setting, the advantages of BP accumulators are apparent in both prover and verification costs. In decentralized identity systems with privacy, issuers sign the users attributes/identity and these signatures (aka the credential) are kept in an accumulator. Later, users should be able to perform batch proofs of (non-)membership for their stored credentials. Typically, the underlying signature schemes output integers (dlog, RSA-based schemes) or group elements that can be trivially mapped to integers (ECC schemes). Thus, one can immediately utilize a BP accumulator. If an RSA accumulator is used to hold the credentials, a $\mathsf{H}_{\mathsf{prime}}$ function (maps to prime) needs to be applied to each element individually and a ZK proof of primality is required. Ozdemir et al. [105] show how the use of Pocklington certificates reduce the cost of proving primality in ZK, but it only reduces the number of Miller-Rabin primality tests. This still results in non-constant verification time. This hashing operation, along with a primality test, needs to also be included in the

ZK proof. This makes the computation of the ZK proof much more complex (e.g., if one uses zk-SNARKs or other specialized ZK proofs for non-algebraic statements [7]).

Limitations. While both BP and RSA accumulators require a trusted setup phase to compute the public parameters, the size of BP parameters is significant: 18 MiB for 2^{17} elements (although not necessarily needed for verification). These parameters grow even more ($3\times$) in our ZK setting due to the use of the knowledge-of-exponent assumption. Finally, it should be noted that in BP accumulators (as opposed to RSA), it is unknown how to add or generate proofs of (non-)membership without the knowledge of the entire accumulated set X.

4.1 Related work

Based on the use of *trapdoors* for accumulator operations, accumulators can be: *trapdoor-based* or *trapdoorless*. In a *trapdoor-based* accumulator, a trusted entity, called the *accumulator manager*, holds some secret trapdoor information, which allows the entity to efficiently perform accumulator operations. A *trapdoorless* accumulator on the other hand, operates without the trapdoor and if a trapdoor is used during setup, it is later destroyed.

We classify prior works into three broad categories: (1) accumulators based on hash functions, (2) accumulators in hidden-order groups, and (3) accumulators in known-order groups.

Hash-based. Accumulators built based on Merkle tree [13,96] or Bloom-filters [158] do not support batching, aggregation, and ZK proof of batch (non-)membership without general purpose tools such as SNARKs, unlike the techniques proposed in this work.

Hidden-order groups. In the single-proof setting, Camenisch and Lysyanskaya [41] proposed the first *dynamic* accumulator (supports both set difference and set union without requiring to fully recompute the accumulator from scratch) secure under *strong RSA* assumption based on prior accumulator constructions [15, 21]. However, their construction is not in the trapdoorless setting as the trapdoor is used to delete elements from the accumulators. Li et al. [92] proposed the first *universal* accumulator (supports both membership and non-membership proofs) by generalizing the Camenisch and Lysyanskaya accumulators [41] to support non-membership proofs under the strong RSA assumption. They also provided efficient algorithms to update non-membership proofs on changes to the accumulated set.

Boneh et al. [26] support batching and aggregation of membership and non-membership proofs in the distributed and trapdoorless setting. They also leverage their contributions to realize a stateless blockchain [26, 50] in the UTXO setting. However, RSA based constructions have larger proof size and verification cost when compared to the BP-based constructions.

Known-order groups. In the single-proof setting, Nguyen proposed the first accumulator [103] using bilinear-maps and based on the t-strong bilinear Diffie-Hellman assumption. In a later work, Damgård et al. [55] and Au et al. [11] extended the accumulator construction by Nguyen to support constant-sized non-membership proof for a *single* element. This non-membership proof scheme relies on Polynomial Remainder Theorem (PRT) to prove non-membership [155]. However, this construction does not extend to a constant-sized batch non-membership proofs or consider efficient aggregation of non-membership proofs. In our work, we study constant-sized batch proofs and present aggregation techniques for

the greatest common divisor (GCD) based non-membership construction, rather than the PRT-based.

Thakur [134] propose batching techniques for BP accumulators (for weak soundness and without rigorous analysis). In their latest version, they also propose two PoE protocols. However, one of the PoE approaches assumes that it is possible to exponentiate an arbitrary base to the trapdoor (which is only feasible in the trapdoor-based setting). Connolly et al. [54] also propose a PoE protocol based on [134], under a different assumption (q-co-GSDH). Compared to [134] and [54], our PoE is concretely one exponentiation fewer under the adaptive variant of q-SDH. Finally, neither [134] nor [54] consider aggregation or ZK batch proofs.

Prior works [45, 54, 69, 110] define a batch proof for multiple elements. However, these works either: (1) assume in their soundness definition that the accumulator is well-formed and honestly computed or (2) does not consider aggregation [69] or (3) rely on an accumulator manager [145]. On the other hand, our batch proofs are sound even for an adversarially chosen accumulator value and our techniques support aggregation in the trapdoorless setting.

Camenisch et al. [40] and Zhang et al. [164] proposed BP accumulators that are algebraically quite different from [103]. These schemes have parameters that are equal to the size of the accumulator domain or more.

Vector Commitment (VC). A VC is a primitive closely related to accumulators, that provides a succinct commitment and positional binding to an *ordered* set of values [46]. Catalano and Fiore proposed a technique to transform a VC into an accumulator [46].

However, this approach results in an accumulator scheme with public parameters that are equal to the size of the accumulator domain [46, 50, 72, 90, 141].

Tomescu et al. proposed aggregatable sub-vector commitments in the bilinear-map setting based on Lagrange polynomials and KZG commitments [82, 141]. Their work uses the partial fraction decomposition technique [153] to aggregate VC proofs. We adopt techniques to aggregate proofs in BP accumulators. Campanelli et al. [42] defined incremental aggregation, i.e., aggregation of aggregated proofs for a RSA-based VC. Although inspired by [26], their work cannot be efficiently directly applied to accumulators.

Zero-knowledge (ZK). There are known techniques to efficiently achieve ZK for a single element in the BP [11] and the RSA [22, 41] setting. Naively, a batch ZK RSA proof corresponds to proving that the exponent is the product of multiple distinct elements, which results in a linear size proof. Recent work by Campanelli et al. [43] constructs ZK proofs of batch membership for RSA using SNARKs and get a constant size proof. Their construction can be transformed to prove non-membership (it corresponds to membership of intervals) but it cannot support both membership and non-membership at the same time without having an accumulator manager holding the trapdoor.

4.2 Preliminaries

Let $\mathbb{Z}_p$ be a field of prime order and $\mathbb{Z}_p[x]$ be a polynomial ring. We denote the degree of polynomial $I(x) \in \mathbb{Z}_p[x]$ as $\mathsf{deg}(I)$. When $\mathbb{G}_1 = \mathbb{G}_2 = \langle g \rangle$, the pairing is called *symmetric* and is denoted as $(p, \mathbb{G}, \mathbb{G}_T, e, g)$.

Partial fraction decomposition (PFD). A rational polynomial can be decomposed into simpler fractions [153]. Concretely, let $A(x) = \prod_{i \in I}(x + a_i)$ be a polynomial and let $A'(x)$ be the first derivative of $A(x)$ with respect to x. Then,

$$\frac{1}{A(x)} = \sum_{i \in I} \frac{1}{A'(-a_i)(x + a_i)}.$$

Polynomial remainder theorem (PRT). When a polynomial $A(x)$ is divided by $(x+r)$, the remainder is the evaluation of $A(x)$ at $-r$. Let $q(x)$ denote the quotient polynomial [155]. Then,

$$A(x) = q(x)(x + r) + A(-r).$$

Bézout's theorem (for polynomials). Given $f(x), g(x) \in \mathbb{Z}_p[x]$, there exists $p(x), q(x), h(x) \in \mathbb{Z}_p[x]$ such that [154]:

$$p(x)f(x) + q(x)g(x) = \gcd(f(x), g(x)) = h(x)$$

Moreover, $\mathsf{deg}(p) < \mathsf{deg}(g) - \mathsf{deg}(h)$ and $\mathsf{deg}(q) < \mathsf{deg}(f) - \mathsf{deg}(h)$.

Pedersen vector commitment (PVC). Given a group $\mathbb{G}_1$ or $\mathbb{G}_2$ of prime order p and a vector $\vec{c} = (c_0, \ldots, c_t) \in \mathbb{Z}_p^t$. Let $(g_1, g_2, \ldots, g_t, h) \in \mathbb{G}_1^{t+1}$ or $\mathbb{G}_2^{t+1}$ generators where $\log_{g_i} g_j, i \neq j$ relationship is unknown. In order to commit to the vector $\vec{c}$, one has to pick $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and compute $\mathsf{PVC}(\vec{c}, r) = h^r g_0^{c_0} g_1^{c_1} \ldots g_t^{c_t}$.

The Pedersen commitment scheme [116] (and its vector generalization) is homomorphic, perfectly hiding and computationally binding under the discrete logarithm assumption.

Zero-knowledge proofs. A ZK proof for a relation $\mathcal{R}(x; w)$, where x is the public statement and w is the witness, is a set of algorithms (Setup, Prove, Verify) with the following syntax:

- $\mathsf{Setup}(1^\lambda, \mathcal{R}) \to \mathsf{pp}$: given the security parameter λ and the relation, outputs parameters pp.
- $\mathsf{Prove}(\mathsf{pp}, x, w) \to \pi$: given the parameters pp, a statement x and a witness w, it returns a proof π for $\mathcal{R}(x; w)$.
- $\mathsf{Verify}(\mathsf{pp}, \mathsf{x}, \pi) \to b \in \{0, 1\}$: given the parameters pp, the statement x and the proof π, it accepts or rejects the proof.

Properties. A ZK proof has to be *correct, sound and zero-knowledge.* Correctness means that if $(x, w) \in \mathcal{R}$, then $\mathsf{Verify}(\mathsf{pp}, \mathsf{x}, \pi) = 1$ with overwhelming probability. We support *knowledge soundness* - if a proof passes verification, then there exists a polynomial time algorithm (the extractor) which by interacting with the prover can extract the witness. Finally, zero-knowledge implies that the proof leaks nothing about the witness, i.e., there exists a simulator with access only to the public statement which can output a valid proof.

4.2.1 Cryptographic accumulators

An accumulator is a cryptographic primitive that supports a succinct binding commitment to an arbitrary set of values. In this work, we consider trapdoorless, dynamic, and universal accumulators. Following the definitions and notation from [13] and [26], our definitions refer to the *batch* setting, where I is a set of elements. The traditional accumulator algorithms can be derived for $I = \{x\}$.

Definition 4.2.1 (Trapdoorless Accumulator)**.** *Let $\mathcal{D}$ be the domain of the accumulated elements, and consider set $X \in \mathcal{D}^{|X|}$ and set $I \in \mathcal{D}^{|I|}$ (where each element from sets X and I is in $\mathcal{D}$). An accumulator consists of the following* PPT *algorithms:*

1. $(\mathsf{pp}, \mathcal{D}) \leftarrow \mathsf{Acc.Setup}(1^\lambda)$: Takes security parameter λ, returns the public parameters pp and $\mathcal{D}$ of the accumulated set. A *trusted* entity may use a secret trapdoor to generate pp. The trapdoor is then destroyed by the setup. In the rest of the document, we assume that $\mathsf{pp} = (\mathsf{pp}, \mathcal{D})$.

2. $A_X \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(X = \{x_1, \ldots, x_{|X|}\})$: Takes set $X \in \mathcal{D}^{|X|}$, outputs the accumulator digest A_X.

3. $A'_X \leftarrow \mathsf{Acc.Add}_{\mathsf{pp}}(A_X, X, I)$: Adds set $I \in \mathcal{D}^{|I|}$, $I \cap X = \emptyset$, to the accumulator and returns the new digest A'_X.

4. $A'_X \leftarrow \mathsf{Acc.Del}_{\mathsf{pp}}(A_X, X, I)$: Removes set I, $I \subseteq X$, from the accumulator and returns the new accumulator value, A'_X.

5. $\pi_I \leftarrow \mathsf{Acc.MemProve}_{\mathsf{pp}}(X, I)$: Generates a membership proof for set I, $I \subseteq X$.

6. $\{0, 1\} \leftarrow \mathsf{Acc.MemVerify}_{\mathsf{pp}}(A_X, I, \pi_I)$: Returns 1 if the membership proof π_I, for the set I, $I \subseteq X$, is valid against the accumulator digest, A_X.

7. $\overline{\pi}_I \leftarrow \mathsf{Acc.NonMemProve}_{\mathsf{pp}}(X, I)$: Generates non-membership proof for the set I, $I \cap X = \emptyset$, disjoint from the accumulated set.

8. $\underline{\{0,1\} \leftarrow \mathsf{Acc.NonMemVerify}_{\mathsf{pp}}(A_X, I, \overline{\pi}_I)}$: Returns 1 if the non-membership proof $\overline{\pi}_I$, of the set I, $I \cap X = \emptyset$, is valid against the accumulator digest, A_X.

9. $\underline{\pi_I \leftarrow \mathsf{Acc.AggMem}_{\mathsf{pp}}(A_X, I, \{\pi_1, \cdots, \pi_{|I|}\})}$: Combines individual membership proofs $\{\pi_1, \cdots, \pi_{|I|}\}$ into a single aggregated proof.

10. $\underline{\overline{\pi}_I \leftarrow \mathsf{Acc.AggNonMem}_{\mathsf{pp}}(A_X, I, \{\overline{\pi}_1, \cdots, \overline{\pi}_{|I|}\})}$: Combines individual non-membership proofs $\{\overline{\pi}_1, \cdots, \overline{\pi}_{|I|}\}$ into a single aggregated proof.

The following algorithms update a (non-)membership proof of a single element after changes to the accumulated set.

11. $\underline{\pi'_y \leftarrow \mathsf{Acc.MemProofUpdOnAdd}_{\mathsf{pp}}(A_X, X, y, \pi_y, I)}$: Updates the membership proof π_y of element y on addition of set I ($y \notin I, X = X \cup I$) to the accumulator.

12. $\underline{\pi'_y \leftarrow \mathsf{Acc.MemProofUpdOnDel}_{\mathsf{pp}}(A_X, A'_X, X, I, y, \pi_y)}$: Updates the membership proof π_y of element y on deletion of set I, $X = X \setminus I$, from the accumulator.

13. $\underline{\overline{\pi}'_y \leftarrow \mathsf{Acc.NonMemProofUpdOnAdd}_{\mathsf{pp}}(A_X, X, y, \overline{\pi}_y, I)}$: Updates the non-membership proof $\overline{\pi}_y$ of element y disjoint from set X on addition of set I ($y \notin I, X = X \cup I$) to the accumulator.

14. $\underline{\overline{\pi}'_y \leftarrow \mathsf{Acc.NonMemProofUpdOnDel}_{\mathsf{pp}}(A_X, A'_X, X, I, y, \overline{\pi}_y)}$: Updates the non-membership proof $\overline{\pi}_y$ of an element y disjoint from set X on deletion of set I ($X = X \setminus I$) from the accumulator.

4.2.2 Correctness and soundness

The basic security property for accumulators is *soundness* (sometimes called *undeniability* [94]) which states that an adversary *cannot* construct an accumulator A and a set I for which both π_I and $\overline{\pi}_I$ are simultaneously valid. Below we state *strong soundness* (also found in [26]), which allows the adversary to create the accumulator *without* revealing the accumulated set X to the challenger.

Definition 4.2.2 (Soundness)**.** *For any* PPT *adversary $\mathcal{A}$, it holds:*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda) \\ (A, I, \pi_I, \overline{\pi}_I) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \mathsf{Acc.MemVerify}_{\mathsf{pp}}(A, I, \pi_I) = 1 \wedge \\ \mathsf{Acc.NonMemVerify}_{\mathsf{pp}}(A, I, \overline{\pi}_I) = 1 \end{array}\right] = \mathsf{negl}(\lambda)$$

Since no trapdoor is needed for algorithms $\mathsf{Acc.Add}_{\mathsf{pp}}$, $\mathsf{Acc.Del}_{\mathsf{pp}}$, $\mathsf{Acc.MemProve}_{\mathsf{pp}}$, $\mathsf{Acc.NonMemProve}_{\mathsf{pp}}$, $\mathcal{A}$ can adaptively update the accumulator and construct honest proofs during the game before coming up with the accumulator value A and proofs $\pi_I, \overline{\pi}_I$.

Definition 4.2.3 ((Non-)Membership Correctness)**.** *Informally, honestly generated proofs will always verify. Let X be the set of all the elements currently in the accumulator.*

Let $I \subseteq X$,

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda) \\ A_X \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(X) \\ \pi_I \leftarrow \mathsf{Acc.MemProve}_{\mathsf{pp}}(X, I) \\ \mathsf{Acc.MemVerify}_{\mathsf{pp}}(A_X, I, \pi_I) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Let $I \cap X = \emptyset$,

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda) \\ A_X \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(X) \\ \overline{\pi}_I \leftarrow \mathsf{Acc.NonMemProve}_{\mathsf{pp}}(X, I) \\ \mathsf{Acc.NonMemVerify}_{\mathsf{pp}}(A_X, I, \overline{\pi}_I) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 4.2.4 (Aggregation Correctness). *Let $I \subseteq X$ and π_i be the membership proof of $x_i \in I$,*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda) \\ A_X \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(X) \\ (\mathsf{Acc.MemVerify}_{\mathsf{pp}}(A_X, I, \pi_I) = 1)_{i \in [1,|I|]} \\ \pi_I \leftarrow \mathsf{Acc.AggMem}_{\mathsf{pp}}(A_X, I, \{\pi_1, \cdots, \pi_{|I|}\}) \\ \mathsf{Acc.MemVerify}_{\mathsf{pp}}(A_X, I, \pi_I) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Let $I \cap X = \emptyset$ and $\overline{\pi}_i$ be the non-membership proof of $y_i \in I$,

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda) \\ A_X \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(X) \\ (\mathsf{Acc.NonMemVerify}_{\mathsf{pp}}(A_X, y_i, \overline{\pi}_i) = 1)_{i \in [1,|I|]} \\ \overline{\pi}_I \leftarrow \mathsf{Acc.AggNonMem}_{\mathsf{pp}}(A_X, I, \{\overline{\pi}_1, \cdots, \overline{\pi}_{|I|}\}) \\ \mathsf{Acc.NonMemVerify}_{\mathsf{pp}}(A_X, I, \overline{\pi}_I) = 1 \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

We defer the proofs and theorem to the full-version of the paper [130].

4.2.3 Accumulator based on bilinear-maps

In this subsection, we review the standard accumulator based on bilinear-pairing (BP) introduced by Nguyen [103] and improved by [55, 106]. Here, we present the classic BP construction that operates on *individual* proofs. In Section 4.3 and Section 4.4 we show how

to *batch* and *aggregate* proofs in the BP setting, and in Section 4.5 we present a technique to efficiently update non-membership proofs.

Let X be the accumulator set and $X(s) = \prod_{x \in X}(s + x)$ be the accumulator polynomial.

$\underline{\mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda)}$: Let $\mathsf{bp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ be the output of $\mathsf{BilGen}(1^\lambda)$. Assume that s is a trapdoor randomly sampled from $\mathbb{Z}_p^*$. The public parameters are $\mathsf{pp} = (\mathsf{bp}, \{g_1^{s^i}, g_2^{s^i} \mid 0 \leq i \leq t\}, \mathcal{D}' = \mathbb{Z}_p)$ where t is the maximum capacity of the accumulator.

$\underline{A_X \leftarrow \mathsf{Acc.Commit}_{\mathsf{pp}}(X = \{x_1, \ldots, x_{|X|}\})}$: The accumulator digest of the set $X = \{x_1, \cdots, x_n\}$, is $A_X = g_1^{X(s)}$, where the polynomial $X(s) = \prod_{x \in X}(s + x)$.

$\underline{A'_X \leftarrow \mathsf{Acc.Add}_{\mathsf{pp}}(A_X, X, I)}$: A set of elements I, $I \cap X = \emptyset$, can be added to the accumulator and the digest can be updated as: $A'_X = g_1^{\prod_{x \in X \cup I}(s+x)} = A_X^{\prod_{x_i \in I}(s+x_i)}$.

$\underline{A'_X \leftarrow \mathsf{Acc.Del}_{\mathsf{pp}}(A_X, X, I)}$: A set of elements $I \subseteq X$ can be deleted from the accumulator and the digest can be updated as: $A'_X = g_1^{\prod_{x \in X \setminus I}(s+x)} = A_X^{1/\prod_{x_i \in I}(s+x_i)}$.

$\underline{\pi_y \leftarrow \mathsf{Acc.MemProve}_{\mathsf{pp}}(X, y)}$: The proof of membership of an element y can be computed as $\pi_y = g_1^{\prod_{x \in X \setminus \{y\}}(s+x)}$, where X is the accumulated set.

$\underline{\{0,1\} \leftarrow \mathsf{Acc.MemVerify}_{\mathsf{pp}}(A_X, y, \pi_y)}$: The membership proof of an element $y \in X$ in the accumulator can be verified by performing the following pairing check: $e(\pi_y, g_2^y g_2^s) = e(A_X, g_2)$.

$\underline{\overline{\pi}_y = (\alpha, g_1^{\beta(s)}) \leftarrow \mathsf{Acc.NonMemProve}_{\mathsf{pp}}(X, y)}$: The non-membership proof of $\{y\} \cap X = \emptyset$ involves computing the Bézout coefficients $\alpha(s)$ and $\beta(s)$ s.t. $\alpha(s) \cdot X(s) + \beta(s) \cdot (s + y) = 1$. As the monomial $(s + y)$ is of degree one, the polynomial $\alpha(s)$ is in fact a *constant*! Thus, the $\overline{\pi}_y = (\alpha, g_1^{\beta(s)}) \in (\mathbb{Z}_p, \mathbb{G}_1)$ is the non-membership proof of element y.

$\underline{\{0,1\} \leftarrow \mathsf{Acc.NonMemVerify}_{\mathsf{pp}}(A_X, y, \overline{\pi}_y)}$: The non-membership proof can be verified by checking $e(A_X, g_2^\alpha) \cdot e(g_1^{\beta(s)}, g_2^s g_2^y) = e(g_1, g_2)$, where $\overline{\pi}_y = (\alpha, g_2^{\beta(s)})$.

$\underline{\pi'_y \leftarrow \mathsf{Acc.MemProofUpdOnAdd}_{\mathsf{pp}}(A_X, X, y, \pi_y, I)}$: The membership proof of an element π_y, can be updated on addition of set I, to the accumulated set, $X = X \cup I$, by computing, $\pi'_y = \pi_y^{\prod_{z \in I}(s+z)}$.

When $I = \{z\}$, $\pi'_y = A_X \cdot \pi_y^{z-y}$ is the updated proof.

$\pi'_y \leftarrow \mathsf{Acc.MemProofUpdOnDel_{pp}}(A_X, A'_X, X, I, y, \pi_y)$: The membership proof π_y can be updated on deletion of set I from the accumulated set, $X = X \setminus I$, by computing, $\pi'_y = \pi_y^{\frac{1}{\prod_{z \in I}(s+z)}}$. When $I = \{z\}$, $\pi'_y = (\frac{\pi_y}{\pi_z})^{\frac{1}{z-y}}$ is the updated proof.

Non-membership in BP accumulator. There are two ways to prove non-membership in the BP acccumulator: (1) based on the PRT [11,55] and (2) based on GCD [106]. The two methods are equivalent, in that they both require $O(|X|)$ field operations. In this work, we focus on the GCD-based method because it allows for efficient batching and aggregation as we show in Section 4.3.2 and Section 4.4.2. Additionally, we present a new algorithm to update individual non-membership proofs Section 4.5.

BP accumulator soundness. The soundness of the BP construction [103] relies on the t-sBDH assumption (see Assumption 2.1.1).

4.3 Batching BP accumulator proofs

4.3.1 Membership

Recall that the membership proof of a single element $y \in X$ is defined as $\pi_y = g_1^{\prod_{x \in X \setminus y}(s+x)}$ and the membership proof of y can be verified by checking $e(\pi_y, g_2^{(s+y)}) = e(A_X, g_2)$, where A_X is the digest of the accumulator.

Batch proof. A batch membership proof is a single succinct proof for a set of elements in the accumulator. A batch proof contains a polynomial in the exponent that includes every element in the accumulator except the monomials terms that correspond to values in the set I.

$\underline{\pi_I \leftarrow \mathsf{Acc.MemProve_{pp}}(X, I)}$: A batch membership proof for set $I \subseteq X$ is computed as follows: $\pi_I = g_1^{\prod_{x_i \in X \setminus I}(s+x_i)}$

$\underline{\{0,1\} \leftarrow \mathsf{Acc.MemVerify_{pp}}(A_X, I, \pi_I)}$: The batch proof can be verified by checking $e(\pi_I, g_2^{\prod_{x_i \in I}(s+x_i)}) = e(A_X, g_2)$.

Asymptotics. The batch membership proof for I consists only of one element in $\mathbb{G}_1$ (as opposed to $|I| \cdot \mathbb{G}_1$ elements, if proved individually). Verifying a batch proof takes $O(|I| \log^2 |I|)$ field operations to compute the coefficients of polynomial $I(s) = \prod_{x_i \in I}(s + x_i)$ using FFT, one $|I|$-sized multi-exponentiation in $\mathbb{G}_2$ to compute $g_2^{I(s)}$ and 2 pairings. However, verifying $|I|$ individual proofs requires $|I|$ individual exponentiations (cannot use fast multi-exponentiations), and $2|I|$ pairings.

4.3.2 Non-membership

A non-membership proof of an element $y \notin X$ leverages the fact that $\gcd(X(s), (s+y)) = 1$. Thus the Bézout coefficients α and $\beta(s)$ satisfy: $\alpha X(s) + \beta(s)(s + y) = 1$. We generalize this observation to prove the non-membership of set $I \cap X = \emptyset$.

Batch proof. We define a batch non-membership proof as a single succinct proof for a set of elements disjoint from the accumulator. Observe that there are no common roots between polynomials $X(s)$ and $I(s) = \prod_{y_i \in I}(s + y_i)$, therefore $\gcd(X(s), I(s)) = 1$. Moreover, the Bézout coefficients $\alpha(s)$ and $\beta(s)$ of $X(s), I(s)$ are non-constant polynomials.

$\overline{\pi}_I \leftarrow \mathsf{Acc.NonMemProve_{pp}}(X, I)$: A batch non-membership proof for a set I, where $I \cap X = \emptyset$, is $\overline{\pi}_I = (g_2^{\alpha(s)}, g_1^{\beta(s)}) \in (\mathbb{G}_2, \mathbb{G}_1)$ such that $\alpha(s) \cdot X(s) + \beta(s) \cdot \prod_{y_i \in I}(s + y_i) = 1$.

$\{0,1\} \leftarrow \mathsf{Acc.NonMemVerify_{pp}}(A_X, I, \overline{\pi}_I)$: The batch proof can be verified by checking $e(A_X, g_2^{\alpha(s)}) \cdot e(g_1^{\beta(s)}, g_2^{\prod_{y_i \in I}(s+y_i)}) = e(g_1, g_2)$, where $\overline{\pi}_I = ({g_2}^{\alpha(s)}, {g_1}^{\beta(s)})$.

Asymptotics. The batch non-membership proof consists of only one element from $\mathbb{G}_1$ and one element from $\mathbb{G}_2$. Verifying a batch proof takes $O(|I| \log^2 |I|)$ field operations to compute the coefficients of polynomial $I(s)$ using FFT, one $|I|$-sized multi-exponentiation in $\mathbb{G}_2$ to compute $g_2^{I(s)}$ and 2 pairings. However, verifying $|I|$ individual proofs requires $|I|$ individual exponentiations (cannot use fast multi-exponentiations) and $2|I|$ pairings. If $e(g_1, g_2)$ is not precomputed, then batch verification and individual verification requires 3 pairings and $2|I| + 1$ pairings, respectively.

4.4 Aggregation

4.4.1 Membership

The BP accumulator based on Nguyen et al. resembles the KZG polynomial commitments [82,103]. As demonstrated in prior works [36,60,141], KZG polynomial commitments can be aggregated using the PFD [153]. We use these techniques to aggregate proofs in the BP accumulator.

$\pi_I \leftarrow \mathsf{Acc.AggMem_{pp}}(A_X, I, \{\pi_1, \cdots, \pi_{|I|}\})$: Let $I \subseteq X$ be the set of elements to be aggregated, $X(s) = \prod_{x \in X}(s + x)$ be the accumulator polynomial of X, $I(s) = \prod_{x_i \in I}(s + x_i)$ be the accumulator polynomial of I, and $X_i(s) = \prod_{x \in X \setminus \{x_i\}}(s + x)$. The goal is to compute

the polynomial $Y(s) = \prod_{x_i \in X \setminus I}(s + x_i)$, which excludes all the monomials that correspond to values in the set I. Using PFD [153]:

$$Y(s) = X(s) \cdot \frac{1}{I(s)} = X(s) \sum_{x_i \in I} \frac{1}{I'(-x_i)(s + x_i)} = \sum_{x_i \in I} \frac{1}{I'(-x_i)} \cdot X_i(s)$$

Thus, the aggregated proof, $\pi_I = {g_1}^{Y(s)} = \prod_{x_i \in I} \pi_i^{c_i}$, where $c_i = \frac{1}{I'(-x_i)}$, I' is the first derivative of I w.r.t s, and π_i is the individual membership proof of element i.

Correctness and soundness. We can see that as long as each individual π_i's are correct, π_I satisfies *aggregation correctness*. Note that soundness of the accumulator (under Definition 4.2.2) is sufficient to argue the soundness of aggregation. Since the resulting proof after aggregation is a batch proof and thus can be verified by running MemVerify.

Asymptotics. In a field with sufficiently many roots of unity, FFT based polynomial interpolation and multi-point evaluation techniques can be used to compute the polynomial $I(s)$ and evaluate $I'(s)$ at all x_i's in $O(|I| \log^2 |I|)$ field operations. Asymptotically, it takes $O(|I| \log^2 |I|)$ operations in $\mathbb{Z}_p$ (to compute all the c_i's) and $|I|$ sized multi-exponentiation in $\mathbb{G}_1$ to compute the aggregated proof.

The complexity of verifying an aggregated proof is the same as verifying a batch proof. Note that verification requires a single multi-exponentiation of size $|I|$ in $\mathbb{G}_2$ to compute ${g_2}^{I(s)}$. However, using the PoE protocol, we can outsource the exponentiation cost to an untrusted prover. We discuss this optimization in Section 4.6.

4.4.2 Non-membership

In this section, we give an algorithm to combine multiple individual non-membership proofs into a single non-membership proof.

$\overline{\pi}_I \leftarrow \mathsf{Acc.AggNonMem}_{\mathsf{pp}}(A_X, I, \{\overline{\pi}_1, \cdots, \overline{\pi}_{|I|}\})$: Let I be a set of elements disjoint from the accumulated set X, $\overline{\pi}_i = (\alpha_i, g_1^{\beta_i(s)}) \in \mathbb{Z}_p \times \mathbb{G}_1$ be the non-membership proof of the element $y_i \in I$, $X(s) = \prod_{x \in X}(s + x)$ be the accumulator polynomial, $I(s) = \prod_{y_i \in I}(s + y_i)$ be the accumulator polynomial of I, and $Y_i(s) = \frac{I(s)}{(s+y_i)}$.

Observe that $\gcd(Y_1(s), Y_2(s), \ldots, Y_{|I|}(s)) = 1$. By generalization of Bézout's identity for polynomials, $\exists$ polynomials $c_i(s)$, s.t.:

$$\sum_{i=1}^{|I|} c_i(s) \cdot Y_i(s) = \gcd(Y_1(s), Y_2(s), \ldots, Y_{|I|}(s)) = 1 \tag{4.1}$$

Lemma 4.4.1. *The Bézout coefficient polynomials, $c_i(s)$, of Eq. (4.1) are non-zero constants.*

Proof. We argue this using PFD [153].

$$1 = I(s) \cdot \frac{1}{I(s)} = I(s) \sum_{i=1}^{|I|} \frac{1}{I'(-y_i)(s + y_i)} = \sum_{i=1}^{|I|} \frac{1}{I'(-y_i)} \cdot Y_i(s)$$

Thus, $c_i(s) = \frac{1}{I'(-y_i)}$, where $I'(s)$ is the derivative of $I(s)$. □

To compute the non-membership proof of I, we need to compute polynomials $\alpha(s)$ and $\beta(s)$ such that:

$$\alpha(s) \cdot X(s) + \beta(s) \cdot \prod_{y_i \in I}(s + y_i) = \gcd(X(s), \prod_{y_i \in I}(s + y_i)) = 1$$

Recall that each individual non-membership proof satisfies:

$$\alpha_i \cdot X(s) + \beta_i(s) \cdot (s + y_i) = 1, \text{where } \mathsf{deg}(\alpha_i) = 0$$

Multiplying by $c_i \cdot Y_i(s)$

$$\alpha_i c_i Y_i(s) X(s) + c_i \beta_i(s)(s + y_i) Y_i(s) = c_i Y_i(s)$$

$$\alpha_i c_i Y_i(s) X(s) + c_i \beta_i(s) I(s) = c_i Y_i(s)$$

Summing over $i = 1$ to $|I|$

$$\sum_{i=1}^{|I|} \alpha_i c_i Y_i(s) X(s) + \sum_{i=1}^{|I|} c_i \beta_i(s) I(s) = \sum_{i=1}^{|I|} c_i \cdot Y_i(s)$$

Using Eq. (4.1)

$$\left(\sum_{i=1}^{|I|} \alpha_i c_i Y_i(s)\right) \cdot X(s) + \left(\sum_{i=1}^{|I|} c_i \beta_i(s)\right) \cdot I(s) = 1$$

Therefore,

$$\alpha(s) = \sum_{i=1}^{|I|} \alpha_i c_i Y_i(s) \qquad \beta(s) = \sum_{i=1}^{|I|} c_i \beta_i(s) \tag{4.2}$$

Thus the non-membership of set I is $\overline{\pi}_I = ({g_2}^{\alpha(s)}, {g_1}^{\beta(s)})$

Correctness and soundness. We can see that as long as each individual $\overline{\pi}_i$'s are correct, $\overline{\pi}_I$ satisfies *aggregation correctness*. The algorithm $\mathsf{AggNonMem}$ always outputs $\overline{\pi}_I$ because terms c_i do not depend on trapdoor s (Lemma 4.4.1), therefore exponentiation by c_i is feasible. Note that soundness of the accumulator (under Definition 4.2.2) is sufficient to argue the soundness of aggregation. Since the resulting proof after aggregation is a batch proof and thus can be verified by running $\mathsf{NonMemVerify}$.

Asymptotics. The task of computing the aggregated proof can be divided into: (1) task of computing the Bézout coefficients c_i's, (2) task of computing the coefficients of each $Y_i(s)$, and (3) task of computing $({g_2}^{\alpha(s)}, {g_1}^{\beta(s)})$.

First, the task of computing the Bézout coefficients in Eq. (4.1) takes $O(|I| \log^2 |I|)$ field operations. Second, the task of computing all Y_i's take $O(|I|^2 \log |I|)$ field operations. This is because computing each $Y_i(s)$ costs $O(|I| \log |I|)$ field operations. Third, the task of computing ${g_2}^{\sum_{i=1}^{|I|} \alpha_i c_i Y_i(s)}$ and ${g_1}^{\sum_{i=1}^{|I|} c_i \beta_i(s)}$ requires a single multi-exponentiation of size $|I|$ in $\mathbb{G}_2$ and $\mathbb{G}_1$, respectively. Thus, in total, it takes $O(|I|^2 \log |I|)$ field operations and multi-exponentiations of size $O(|I|)$ in $\mathbb{G}_1, \mathbb{G}_2$ to compute the aggregated non-membership proof from individual proofs.

The complexity of verifying an aggregated proof is the same as verifying a batch proof. Note that verification requires a single multi-exponentiation of size $|I|$ in $\mathbb{G}_2$ to compute ${g_2}^{I(s)}$. However, using the PoE protocol, we can outsource the exponentiation cost to an untrusted prover. We discuss this optimization in Section 4.6. We also illustrate the non-membership aggregation technique using an example in Section 4.9.

Scheme	RSA	BP
Domain	Prime	$\mathbb{Z}_p$
$\mathsf{Setup}(1^\lambda)$	$O(1)$	$O(t) \cdot (\mathbb{G}_1 + \mathbb{G}_2)$
Commit	$O(\lvert X \rvert) \cdot \mathbb{G}$	$O(\lvert X \rvert) \cdot \mathbb{G}_1 +$ $O(\lvert X \rvert \log^2 \lvert X \rvert) \cdot \mathbb{Z}_p$
AggMem	$O(\lvert I \rvert \log \lvert I \rvert) \cdot \mathbb{G} +$ $O(\lvert I \rvert \log \lvert I \rvert) \cdot \mathbb{F}$	$O(\lvert I \rvert) \cdot \mathbb{G}_1 +$ $O(\lvert I \rvert \log^2 \lvert I \rvert) \cdot \mathbb{Z}_p$
MemVerify	$O(\lvert I \rvert) \cdot \mathbb{G}$	$2\mathbb{P} + O(\lvert I \rvert) \cdot \mathbb{G}_2 +$ $O(\lvert I \rvert \log^2 \lvert I \rvert) \cdot \mathbb{Z}_p$
MemVerifyPoE	$O(1) \cdot \mathbb{G} +$ $O(\lvert I \rvert) \cdot \mathbb{F}$	$3\mathbb{P} + 1\mathbb{G}_1 + 1\mathbb{G}_2 +$ $O(\lvert I \rvert \log^2 \lvert I \rvert) \cdot \mathbb{Z}_p$
AggNonMem	$O(\lvert I \rvert \log \lvert I \rvert) \cdot \mathbb{G} +$ $O(\lvert I \rvert \log \lvert I \rvert) \cdot \mathbb{F}$	$O(\lvert I \rvert) \cdot (\mathbb{G}_1 + \mathbb{G}_2) +$ $O(\lvert I \rvert^2 \log \lvert I \rvert) \cdot \mathbb{Z}_p$
NonMemVerify	$O(\lvert I \rvert) \cdot \mathbb{G}$	$2\mathbb{P} + O(\lvert I \rvert) \cdot \mathbb{G}_2 +$ $O(\lvert I \rvert \log^2 \lvert I \rvert) \cdot \mathbb{Z}_p$
NonMemVerifyPoE	$O(1) \cdot \mathbb{G} +$ $O(\lvert I \rvert) \cdot \mathbb{F}$	$5\mathbb{P} + 1\mathbb{G}_1 + 1\mathbb{G}_2 +$ $O(\lvert I \rvert \log^2 \lvert I \rvert) \cdot \mathbb{Z}_p$

Table 4.1: Set X denotes the entire accumulated set and $I \subseteq X$. Let $O(Y) \cdot \mathbb{G}$ denotes one large exponentiation to the product of Y elements or Y exponentiations. All exponentiations in BP can be sped up by a logarithmic factor using multi-exponentiations.

4.5 Non-Membership proof updates

In this section, we describe a new protocol that allows to efficiently update a GCD-based non-membership proof for the BP accumulator after changes (additions/deletions) to the accumulated set. Let $y \notin X$ and $\overline{\pi}_y = (\alpha, {g_1}^{\beta(s)}) \in \mathbb{Z}_p \times \mathbb{G}_1$ s.t. $\alpha X(s) + \beta(s)(s+y) = 1$.

$\underline{\overline{\pi}'_y \leftarrow \mathsf{Acc.NonMemProofUpdOnAdd}_{\mathsf{pp}}(A_X, X, y, \overline{\pi}_y, I = \{z\})}$: Recall that,

$$\alpha X(s) + \beta(s)(s+y) = 1 \tag{4.3}$$

Since $y \neq z$, by Bézout's Identity,

$$u(s+z) + v(s+y) = 1, \text{ where } u, v \in \mathbb{Z}_p \tag{4.4}$$

Goal is to find α' and $\beta'(s)$ s.t.,

$$\alpha' X'(s) + \beta'(s)(s+y) = 1, \text{ where } X'(s) = X(s)(s+z) \tag{4.5}$$

Multiplying Eq. (4.3) by $u(s+z)$

$$u\alpha X(s)(s+z) + u\beta(s)(s+z)(s+y) = u(s+z)$$

Using Eq. (4.4):

$$u\alpha X(s)(s+z) + u\beta(s)(s+z)(s+y) = 1 - v(s+y)$$

$$(u\alpha)X'(s) + (v + u\beta(s)(s+z))(s+y) = 1$$

From Eq. (4.5) we have: $\alpha' = u\alpha$ and $\beta'(s) = v + u\beta(s)(s+z)$

However, it is not possible to compute ${g_1}^{\beta'(s)}$ without the coefficients of $\beta(s)$ because individual proofs only contain $g^{\beta(s)}$. Thus, we simplify $\beta'(s)' = v + u\beta(s)(s+z)$, replace $u(s+z)$ as $1-v(s+y)$ from Eq. (4.4): $\beta'(s) = v+\beta(s)(1-v(s+y)) = v+\beta(s)-v\beta(s)(s+y)$ $= v(1-\beta(s)(s+y)) + \beta(s)$. Replace $1-\beta(s)(s+y)$ as $\alpha X(s)$ from Eq. (4.3): $\beta'(s) = v\alpha X(s) + \beta(s)$.

Thus, $\alpha' = u\alpha$, $\beta'(s) = v\alpha X(s) + \beta(s)$ and $\overline{\pi}'_y = (\alpha', {g_1}^{\beta'(s)}) = (u\alpha, {A_X}^{v\alpha} \cdot {g_1}^{\beta(s)})$.

To compute the updated non-membership proof, $\overline{\pi}'_y = (\alpha', {g_1}^{\beta'(s)})$, we calculate constants u, v in Eq. (4.4) by running the extended Euclidean algorithm for degree one polynomials, which takes only $O(1)$ operations. We can compute the constant α' with one multiplication operation in $\mathbb{Z}_p$. Similarly, we can compute ${g_1}^{\beta'(s)} = ((A_X)^{\alpha})^{v} \cdot {g_1}^{\beta(s)}$ with one multiplication operation in $\mathbb{Z}_p$, one exponentiation operation in $\mathbb{G}_1$ by $\mathbb{Z}_p$, and one addition in $\mathbb{G}_1$.

$\underline{\overline{\pi}'_y \leftarrow \textsf{Acc.NonMemProofUpdOnDel}_{\textsf{pp}}(A_X, A'_X, X, I = \{z\}, y, \overline{\pi}_y)}$: Recall that, $\alpha X(s)+\beta(s)(s+y) = 1$. Since $y \neq z$, by Bézout's Identity,

$$u(s+z) + v(s+y) = 1, \text{ where } u, v \in \mathbb{Z}_p \tag{4.6}$$

Goal is to find α' and $\beta'(s)$ s.t.,

$$\alpha' X'(s) + \beta'(s)(s+y) = 1, \text{ where } X'(s) = \frac{X(s)}{(s+z)} \tag{4.7}$$

$$\alpha X(s) + \beta(s)(s+y) = 1$$

$$\alpha X'(s)(s+z) + \beta(s)(s+y) = 1$$

Replace $(s+z)$ as $\frac{1-v(s+y)}{u}$ from Eq. (4.6):

$$\alpha X'(s)\left(\frac{1-v(s+y)}{u}\right) + \beta(s)(s+y) = 1$$

$$\frac{\alpha}{u}X'(s) + \left(\beta(s) - \frac{v\alpha}{u}X'(s)\right)(s+y) = 1$$

Thus, from Eq. (4.7), $\alpha' = \frac{\alpha}{u}$, $\beta'(s) = \beta(s) - v\alpha' X'(s)$ and $\overline{\pi}'_y = (\alpha', {g_1}^{\beta'(s)}) = \left(\frac{\alpha}{u}, \frac{{g_1}^{\beta(s)}}{A_{X'}^{v\alpha'}}\right)$.

To compute the updated non-membership proof, $\overline{\pi}'_y = (\alpha', {g_1}^{\beta'(s)})$, we calculate constants u, v in Eq. (4.6) by running the extended Euclidean algorithm for degree one polynomials, which takes only $O(1)$ operations. We can compute the constant α' with one inversion operation and one multiplication operation in $\mathbb{Z}_p$. Similarly, we can compute ${g_1}^{\beta'(s)} = \frac{{g_1}^{\beta(s)}}{A_{X'}^{v\alpha'}}$ with one multiplication operation in $\mathbb{Z}_p$, one inversion operation, one exponentiation operation in $\mathbb{G}_1$ by $\mathbb{Z}_p$, and one addition operation in $\mathbb{G}_1$.

4.6 Proof of Exponentiation

We present the Proof-of-Exponentiation (PoE) protocol in the known-order group setting. Informally, the prover can convince the verifier that the exponentiation of a group element by the evaluation of a known polynomial at a specific point is correct. That is, given a tuple, $(A_U, A_W, V(x)) \in (\mathbb{G}, \mathbb{G}, \mathbb{Z}_p[x])$, the prover can convince the verifier that $A_W = A_U^{V(s)}$, with a constant sized proof requiring constant number of pairing checks. Given the coefficients of polynomial $V(s)$, naively, computing $g^{V(s)}$ would require as many exponentiations as $\mathsf{deg}(V(s))$. However, with PoE, the verifier has to just perform cheaper polynomial division and *constant* pairing computation instead of performing linear number of group exponentiations.

Since naively verifying batch (non-)membership proofs require a multi-exponentiation of size $|I|$, we can delegate the expensive exponentiations to the prover using the PoE protocol. We elaborately discuss this optimization in this section. Note that the PoE protocol is of independent interest and can be used as a building block in other constructions.

For ease of exposition, we present the protocol (Fig. 4.1) using symmetric pairings. However, the protocol can be instantiated using asymmetric pairings.

$$\mathcal{R}_{\mathsf{PoE}} = \left\{ \; ((A_U, A_W \in \mathbb{G}, V(x) \in \mathbb{Z}_p[x]); \perp) : A_W = A_U^{V(s)} \in \mathbb{G} \; \right\}$$

We present the interactive version of the protocol for the symmetric pairing in Fig. 4.1, which can be made non-interactive using Fiat-Shamir transformation. The soundness of $\mathcal{R}_{\mathsf{PoE}}$ is defined similar to its RSA counterparts as defined in [151] and [26]. We defer the proof of soundness of our PoE protocol to Section 4.6.1.

$\underline{\mathsf{pp} \leftarrow \mathsf{PoE.Setup}(1^\lambda)}$:

1. $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^\lambda)$
2. $s \leftarrow\$ \mathbb{Z}_p^*$
3. $\mathsf{pp} := ((p, g, \mathbb{G}, \mathbb{G}_T, e), \{g^{s^i} \mid 0 \leq i \leq t\})$

Protocol PoE for $\mathcal{R}_{\mathsf{PoE}}$:
Params: $\mathsf{pp} \leftarrow \mathsf{PoE.Setup}(1^\lambda)$
Inputs: $(A_U, A_W \in \mathbb{G}, V(x) \in \mathbb{Z}_p[x])$
Claim: $A_W = A_U^{V(s)} \in \mathbb{G}$

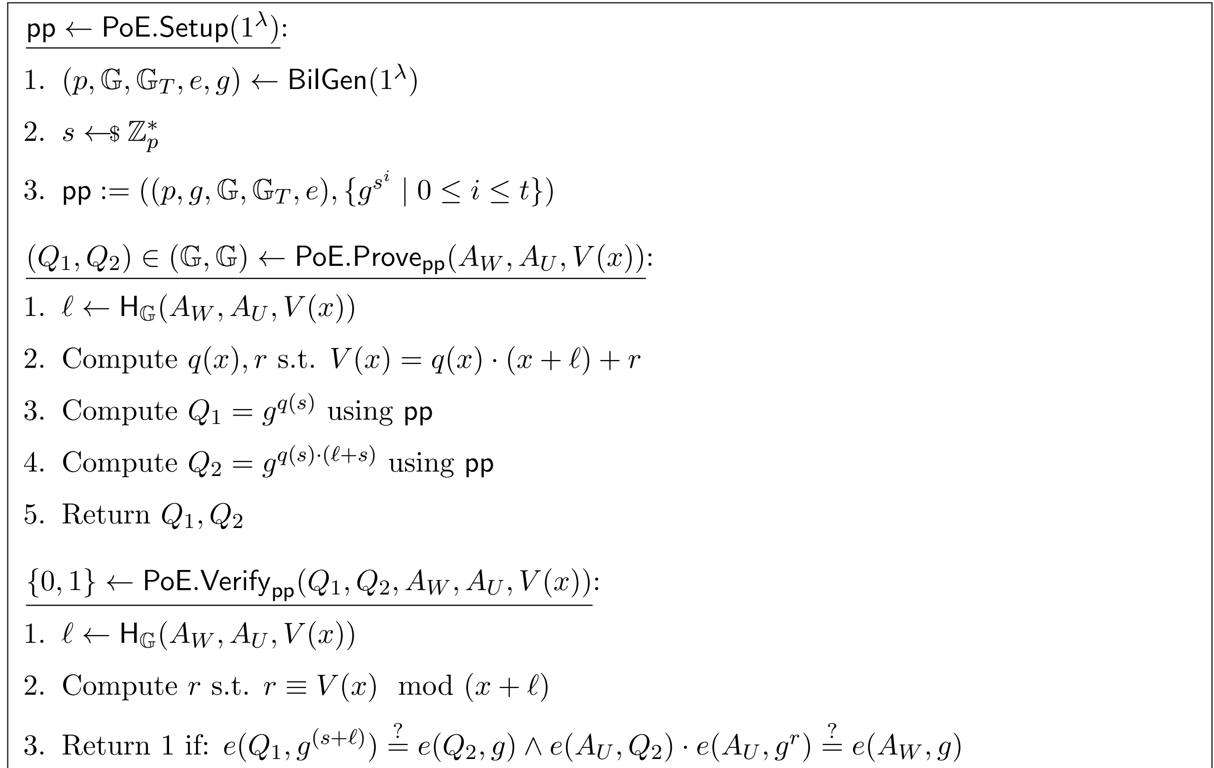
1. Verifier sends $\ell \leftarrow\$ \mathbb{Z}_p$
2. Prover computes:
   $q(x), r$ s.t. $V(x) = q(x) \cdot (x + \ell) + r$
   $Q_1 = g^{q(s)}$ using pp
   $Q_2 = g^{q(s)\cdot(\ell+s)}$ using pp
3. Prover sends Q_1, Q_2 to Verifier.
4. Verifier computes: r s.t. $r \equiv V(x) \mod (x + \ell)$
   Accepts if: $e(Q_1, g^{(s+\ell)}) \stackrel{?}{=} e(Q_2, g) \wedge e(A_U, Q_2) \cdot e(A_U, g^r) \stackrel{?}{=} e(A_W, g)$

Figure 4.1: PoE protocol. We use Fiat-Shamir transformation to make this protocol into non-interactive (Fig. 4.2). For the ease of exposition we present the construction in the symmetric pairing setting. However, we remark that our implementation uses asymmetric paring.

Membership proof aggregation. Recall that (from Section 4.4.1) to verify an aggregated membership proof, the verifier has to compute ${g_2}^{I(s)} = {g_2}^{\prod_{x_i \in I}(s+x_i)}$ and perform two pairing computations. Computing ${g_2}^{I(s)}$, naively, requires $O(|I| \log^2 |I|)$ operations to compute the coefficients of the polynomial $I(s)$, and $O(|I|)$ exponentiations to compute ${g_2}^{I(s)}$. However, with PoE, the verifier can delegate the computation ${g_2}^{I(s)}$ to an untrusted prover (we can set the group $\mathbb{G}$ in PoE to equal to $\mathbb{G}_2$ from the accumulator setup phase). Thus, with PoE, the verifier now incurs the cost of polynomial division of $I(s)$ by a monomial and constant pairing computations instead of computing $O(|I|)$ exponentiations. Note that the

PoE protocol is sound for an arbitrary element $A_U \in \mathbb{G}_2$, thus the soundness also holds for g_2.

Rather than delegating the computation of ${g_2}^{I(s)}$, the verifier can altogether delegate the computation of ${\pi_I}^{I(s)} = A_X$ to an untrusted prover using the PoE protocol, as it can support arbitrary $A_U \in \mathbb{G}_1$. Thus the verifier, in addition, to trading the exponentiation costs for polynomial division, also saves a pairing computation when compared to delegating the computation of ${g_2}^{I(s)}$.



$\underline{\mathsf{pp} \leftarrow \mathsf{PoE.Setup}(1^\lambda)}$:

1. $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^\lambda)$
2. $s \leftarrow\!\$\ \mathbb{Z}_p^*$
3. $\mathsf{pp} := ((p, g, \mathbb{G}, \mathbb{G}_T, e), \{g^{s^i} \mid 0 \leq i \leq t\})$

$\underline{(Q_1, Q_2) \in (\mathbb{G}, \mathbb{G}) \leftarrow \mathsf{PoE.Prove}_{\mathsf{pp}}(A_W, A_U, V(x))}$:

1. $\ell \leftarrow \mathsf{H}_{\mathbb{G}}(A_W, A_U, V(x))$
2. Compute $q(x), r$ s.t. $V(x) = q(x) \cdot (x + \ell) + r$
3. Compute $Q_1 = g^{q(s)}$ using pp
4. Compute $Q_2 = g^{q(s)\cdot(\ell+s)}$ using pp
5. Return Q_1, Q_2

$\underline{\{0,1\} \leftarrow \mathsf{PoE.Verify}_{\mathsf{pp}}(Q_1, Q_2, A_W, A_U, V(x))}$:

1. $\ell \leftarrow \mathsf{H}_{\mathbb{G}}(A_W, A_U, V(x))$
2. Compute r s.t. $r \equiv V(x) \mod (x + \ell)$
3. Return 1 if: $e(Q_1, g^{(s+\ell)}) \stackrel{?}{=} e(Q_2, g) \wedge e(A_U, Q_2) \cdot e(A_U, g^r) \stackrel{?}{=} e(A_W, g)$

Figure 4.2: Non-interactive PoE protocol.

Non-membership proof aggregation. Similarly, recall that (from Section 4.4.2) to verify an aggregated non-membership proof, the verifier has to compute ${g_2}^{I(s)} = {g_2}^{\prod_{y_i \in I}(s+y_i)}$ and perform two pairing computations. Similarly, with PoE, the verifier can delegate the computation ${g_2}^{I(s)}$ to an untrusted prover. Thus, with PoE, the verifier now incurs the cost

of polynomial division of $I(s)$ by a linear polynomial and constant pairing computation instead of computing $O(|I|)$ exponentiations to verify the nonmembership proof.

4.6.1 Soundness of PoE

Theorem 4.6.1. *Under the t-Strong bilinear adaptive Diffie-Hellman assumption (Assumption 2.1.2),* PoE *is an argument system for relation* $\mathcal{R}_{\mathsf{PoE}}$ *with soundness error,* $\mathsf{negl}(\lambda)$.

Proof. Assume that the adversary outputs Q_1, Q_2 s.t. $A_W \neq A_U^{V(s)}$. And assume that this tuple is accepted by the verifier. This implies that:

$$
\begin{aligned}
e(A_U, {Q_1}^{(\ell+s)}) \cdot e(A_U, g^r) &= e(A_W, g) \\
e(A_U, {Q_1}^{(\ell+s)}) \cdot e(A_U, g^{V(s)-q(s)\cdot(\ell+s)}) &= e(A_W, g) \\
e(A_U, {Q_1}^{(\ell+s)}) \cdot e(A_U, g^{V(s)}) \cdot e(A_U, g^{-q(s)\cdot(\ell+s)}) &= e(A_W, g) \\
e(A_U, {Q_1}^{(\ell+s)}) \cdot e(A_U, g^{-q(s)\cdot(\ell+s)}) &= e(A_W, g) \cdot e(A_U, g^{-V(s)})
\end{aligned}
$$

Taking root $(\ell + s)$

$$
\begin{aligned}
e(A_U, Q_1) \cdot e(A_U, g^{-q(s)}) &= [e(A_W, g) \cdot e(A_U, g^{-V(s)})]^{\frac{1}{(\ell+s)}} \\
&= e(A_W \cdot A_U^{-V(s)}, g)^{\frac{1}{(\ell+s)}}
\end{aligned}
$$

Since $A_W \cdot A_U^{-V(s)} \neq 1$. Thus $\mathcal{A}$ computes $e(w, g)^{\frac{1}{(\ell+s)}}$, which breaks Assumption 2.1.2. □

4.7 ZK batch proofs

We describe the zero-knowledge batch (non-)membership proofs that is experimentally evaluated in this work. We consider the following setting: a set $X = \{x_1, \ldots, x_n\}$ of elements is accumulated, and a prover holds witnesses for a subset $I \subseteq X$, where $|I| = d$. The goal is to prove that $I \subseteq X$ (or $I \cap X = \emptyset$) while *hiding the set* I itself. Additionally, reveal just the size of I or in other words that it includes "at least d elements". This is important since typically a batch proof does not hide the number of batched elements and this is useful in identity systems, sanctions/embargoed lists, e-cash, etc. At the same time, the goal is to maintain the benefits of batch proofs: (1) the verifier cost should be constant for both membership and non-membership proofs, and (2) the size of the ZK batch proof should remain sublinear.

In the ZK setting, the verifier does not hold the set I or the proof π_I (or $\overline{\pi}_I$) in order to run the verification algorithm. Instead, the prover has to prove in ZK that the pairing equations in the verification algorithm hold. The prover's witness is the set I, the randomness r used in the commitment to I, a proof of membership π_I or a proof of non-membership $\overline{\pi}_I$. The inputs known to the verifier are the public parameters pp, a commitment to the subset C_I, and the accumulator value A_X.

More specifically, the prover (that knows the polynomial $I(x)$ and the commitment randomness r) has to compute ZK proofs for the following relations:

- $\mathcal{R}_{\mathsf{mem}}(\mathsf{pp}, C_I, A_X; \pi_I, r)$ knowledge of π_I such that membership verification holds and knowledge of randomness r

- $\mathcal{R}_{\mathsf{nonmem}}(\mathsf{pp}, C_I, A_X; \overline{\pi}_I, r)$ knowledge of $\overline{\pi}_I$ such that non-membership verification holds and knowledge of randomness r

- If revealing the size, $\mathcal{R}_{\mathsf{degcheck}}(\mathsf{pp}, C_I, d; I, r)$ proves that the set size $|I| = d$ and the set corresponds to C_I. A proof for this relation is analyzed in the following tasks:

  – Proving that $I(x) = \prod_{i=1}^{d}(x + x_i) = x^d + f(x)$, where $\mathsf{deg}(f) \leq d - 1$ using a hiding commitment C_f,

  – Proving well-formedness of C_f in relation to C_I.

The commitment C_I allows the prover to later open all elements in I (if needed) by revealing r. In the following paragraphs, we explain in details how these proofs are constructed.

Notation. Let pp be the following public parameters:

- $g_1, h_1, \mathfrak{g}, \mathfrak{g}_1, \mathfrak{g}_2 \in \mathbb{G}_1$ and $g_2, h_2, \mathfrak{h}, \mathfrak{h}_1, \mathfrak{h}_2 \in \mathbb{G}_2$

- $\mathbf{g_1^s} = [g_1, g_1^s, \ldots, g_1^{s^t}] \in \mathbb{G}_1^{t+1}$, $\mathbf{g_2^s} = [g_2, g_2^s, \ldots, g_2^{s^t}] \in \mathbb{G}_2^{t+1}$

- $\mathbf{a} = [g_2^a, g_2^{as}, \ldots, g_2^{as^t}] \in \mathbb{G}_2^{t+1}$

We view a polynomial $I(x)$ as equivalent to its coefficients that form a vector, therefore for the rest of the chapter, we overload the Pedersen VC input with both vectors and polynomials.

4.7.1 Proving membership ($\mathcal{R}_{\mathsf{mem}}$)

The high level idea of the initial proof is the following: $\mathcal{R}_{\mathsf{mem.init}}$ essentially proves that the verification equation holds for $C_I h_2^{-r}$ and proves knowledge of r.

$$\mathcal{R}_{\mathsf{mem.init}} = \left\{ \begin{array}{c} (\mathsf{pp}, C_I \in \mathbb{G}_2, A_X \in \mathbb{G}_1; \pi_I \in \mathbb{G}_1, r \in \mathbb{Z}_p) : \\ C_I = h_2^r g_2^{I(s)} \wedge e(\pi_I, C_I) \cdot e(\pi_I, h_2)^{-r} = e(A_X, g_2) \end{array} \right\}$$

However, it still does not hide the witness π_I. Using techniques from RingCT [133], $\mathcal{R}_{\mathsf{mem.init}}$ is transformed into $\mathcal{R}_{\mathsf{mem}}$, which proves the above statement in ZK. Specifically, it adds a proof that $\mathcal{R}_{\mathsf{mem.init}}$ verifies for $\pi_{I,2} g^{-\tau_1}$ and proves knowledge of τ_1, where $\pi_{1,2}$ is the blinded version of the batch proof π_I.

$$\mathcal{R}_{\mathsf{mem}} = \left\{ \begin{array}{c} (\mathsf{pp}, C_I \in \mathbb{G}_2, A_X \in \mathbb{G}_1; \pi_I \in \mathbb{G}_1, r, \tau_1, \tau_2 \in \mathbb{Z}_p) : \\ C_I = h_2^r g_2^{I(s)} \wedge \delta_1 = \tau_1 r \wedge \delta_2 = \tau_2 r \wedge \\ \pi_{I,1} = g_1^{\tau_1} \mathfrak{g}^{\tau_2} \wedge \pi_{I,2} = \pi_I \mathfrak{g}^{\tau_1} \wedge \\ \frac{e(\pi_{I,2}, C_I)}{e(A_X, g_2)} = e(\mathfrak{g}, C_I)^{\tau_1} \cdot e(\mathfrak{g}, h_2)^{-\delta_1} \cdot e(\pi_{I,2}, h_2)^r \end{array} \right\}$$

$\mathcal{R}_{\mathsf{mem}}$ is instantiated with a generalized version of Schnorr's protocol for knowledge of discrete log as r, τ_1 are scalars.

Protocol for relation $\mathcal{R}_{\mathsf{mem}}$. The interactive version of the protocol is as follows:

- Prover
  - Picks $\tau_1, \tau_2 \leftarrow \mathbb{Z}_p$
  - Computes $\pi_{I,1} = g_1^{\tau_1} \mathfrak{g}^{\tau_2}$ and $\pi_{I,2} = \pi_I \mathfrak{g}^{\tau_1}$
  - Picks $r_r, r_{\tau_1}, r_{\tau_2}, r_{\delta_1}, r_{\delta_2} \in \mathbb{Z}_p$
  - Sends:
    * $\pi_{I,1}, \pi_{I,2}$
    * $R_1 = g_1^{r_{\tau_1}} \mathfrak{g}^{r_{\tau_2}}$, $R_2 = \pi_{I,1}{}^{r_r} g_1^{-r_{\delta_1}} \mathfrak{g}^{-r_{\delta_2}}$
    * $R_3 = e(\mathfrak{g}, C_I)^{r_{\tau_1}} e(\mathfrak{g}, h_2)^{-r_{\delta_1}} e(\pi_{I,2}, h_2)^{r_r}$
- Verifier sends $c \leftarrow \mathbb{Z}_p$
- Prover sends:

– $s_r = r_r + cr$

– $s_{\tau_1} = r_{\tau_1} + c\tau_1$, $s_{\tau_2} = r_{\tau_2} + c\tau_2$

– $s_{\delta_1} = r_{\delta_1} + c\delta_1$, $s_{\delta_2} = r_{\delta_2} + c\delta_2$

- Verifier checks:

  – $R_1 = \pi_{I,1}^{-c} g_1^{s_{\tau_1}} \mathfrak{g}^{s_{\tau_2}}$

  – $R_2 = \pi_{I,1}^{s_r} g_1^{-s_{\delta_1}} \mathfrak{g}^{-s_{\delta_2}}$

  – $R_3 \cdot (\frac{e(\pi_{I,2}, C_I)}{e(A_X, g_2)})^c = e(\mathfrak{g}, C_I)^{s_{\tau_1}} e(\mathfrak{g}, h_2)^{-s_{\delta_1}} e(\pi_{I,2}, h_2)^{s_r}$

Asymptotics. The instantiation of $\mathcal{R}_{\mathsf{mem}}$ consists of 5 group elements and 5 field elements, and has constant prover and verifier.

4.7.2 Proving non-membership ($\mathcal{R}_{\mathsf{nonmem}}$)

For non-membership, $\mathcal{R}_{\mathsf{mem.init}}$ is replaced by

$$\mathcal{R}_{\mathsf{nonmem.init}} = \left\{ \begin{array}{c} (\mathsf{pp}, C_I \in \mathbb{G}_2, A_X \in \mathbb{G}_1; (\overline{A}, \overline{B}) \in \mathbb{G}_2 \times \mathbb{G}_1, r \in \mathbb{Z}_p) : \\ C_I = h_2^r g_2^{I(s)} \wedge e(A_X, \overline{A}) \cdot e(\overline{B}, C_I) \cdot e(\overline{B}, h)^{-r} = e(g_1, g_2) \end{array} \right\}$$

Similarly, $\mathcal{R}_{\mathsf{nonmem.init}}$ still does not hide the witness π_I. Thus, using techniques from RingCT [133], $\mathcal{R}_{\mathsf{nonmem.init}}$ is transformed into $\mathcal{R}_{\mathsf{nonmem}}$, which proves the above statement in ZK.

$$\mathcal{R}_{\mathsf{nonmem}} = \left\{ \begin{array}{c} (\mathsf{pp}, C_I \in \mathbb{G}_2, A_X \in \mathbb{G}_1; (\overline{A}, \overline{B}) \in \mathbb{G}_2 \times \mathbb{G}_1, r, \tau_1, \tau_3, \tau_4 \in \mathbb{Z}_p) : \\ C_I = h_2^r g_2^{I(s)} \wedge \delta_3 = \tau_3 r \wedge \delta_4 = \tau_4 r \wedge \overline{A}_2 = \overline{A}\mathfrak{h}^{\tau_1} \wedge \\ \overline{B}_1 = g_1^{\tau_3}\mathfrak{g}^{\tau_4} \wedge \overline{B}_2 = \overline{B}\mathfrak{g}^{\tau_3} \wedge \\ \frac{e(A_X, \overline{A}_2) \cdot e(\overline{B}_2, C_I)}{e(g_1, g_2)} = e(A_X, \mathfrak{h})^{\tau_1} \cdot e(\mathfrak{g}, C_I)^{\tau_3} \cdot e(\mathfrak{g}, h_2)^{-\delta_3} \cdot e(\overline{B}_2, h_2)^r \end{array} \right\}$$

The rest of the protocol remains the same.

Protocol for relation $\mathcal{R}_{\mathsf{nonmem}}$. The interactive version of the protocol is as follows:

- Prover
  - Picks $\tau_1, \tau_3, \tau_4 \leftarrow\$ \mathbb{Z}_p$
  - Computes
    * $\overline{A}_2 = \overline{A}\mathfrak{h}^{\tau_1}$
    * $\overline{B}_1 = g_1^{\tau_3}\mathfrak{g}^{\tau_4}$, $\overline{B}_2 = \overline{B}\mathfrak{g}^{\tau_3}$
  - Picks $r_r, r_{\tau_1}, r_{\tau_3}, r_{\tau_4}, r_{\delta_3}, r_{\delta_4} \leftarrow\$ \mathbb{Z}_p$
  - Sends:
    * $\overline{A}_2$, $\overline{B}_1$, $\overline{B}_2$
    * $R_{2,1} = g_1^{r_{\tau_3}}\mathfrak{g}^{r_{\tau_4}}$, $R_{2,2} = (\overline{B}_1)^{r_r} g_1{}^{-r_{\delta_3}}\mathfrak{g}^{-r_{\delta_4}}$
    * $R_3 = e(A_X, \mathfrak{h})^{r_{\tau_1}} \cdot e(\mathfrak{g}, C_I)^{r_{\tau_3}} \cdot e(\mathfrak{g}, h_2)^{-r_{\delta_3}} \cdot e(\overline{B}_2, h_2)^{r_r}$
- Verifier sends $c \leftarrow \mathbb{Z}_p$
- Prover sends:
  - $s_r = r_r + cr$, $s_{\tau_1} = r_{\tau_1} + c\tau_1$
  - $s_{\tau_3} = r_{\tau_3} + c\tau_3$, $s_{\tau_4} = r_{\tau_4} + c\tau_4$
  - $s_{\delta_3} = r_{\delta_3} + c\delta_3$, $s_{\delta_4} = r_{\delta_4} + c\delta_4$
- Verifier checks:
  - $R_{2,1} = (\overline{B}_1)^{-c} g_1{}^{s_{\tau_3}}\mathfrak{g}^{s_{\tau_4}}$
  - $R_{2,2} = (\overline{B}_1)^{s_r} g_1{}^{-s_{\delta_3}}\mathfrak{g}^{-s_{\delta_4}}$
  - $R_3 \cdot (\frac{e(A_X, \overline{A}_2) \cdot e(\overline{B}_2, C_I)}{e(g_1, g_2)})^c = e(A_X, \mathfrak{h})^{s_{\tau_1}} \cdot e(\mathfrak{g}, C_I)^{s_{\tau_3}} \cdot e(\mathfrak{g}, h_2)^{-s_{\delta_3}} \cdot e(\overline{B}_2, h_2)^{s_r}$

4.7.3 Proving degree bound ($\mathcal{R}_{\mathsf{degcheck}}$)

To prove in ZK the following statement: *The prover knows at least d elements.* So far, the verifier only knows that the verification algorithm holds for some commitment C_I. In order to be convinced that C_I is a commitment to a set $I = \{x_1, \ldots, x_d\}$, the verifier has to prove knowledge of a polynomial $I(x)$ of degree d. It is implied that as long as

$I(x)$ is a polynomial, it is well-formed as a product of d monomials $(x + x_i)$ ($I(x)$ is part of the accumulator exponent (membership) or coprime to the accumulator exponent(non-membership)). We define the following relation:

$$\mathcal{R}_{\mathsf{degcheck}} = \left\{ \ (\mathsf{pp}, C_I \in \mathbb{G}_2, d \in \mathbb{Z}_p; r \in \mathbb{Z}_p, I \subset \mathbb{Z}_p) : C_I = h_2^r g_2^{I(s)} \wedge |I| = d \ \right\}$$

For proving the above statement with existing protocols (that prove maximum instead of minimum polynomial degree), it is proved as follows: a correct computation of the polynomial $I(x)$ that corresponds to elements in the set I with $|I| = d$, results to the following polynomial:

$$I(x) = \prod_{i=1}^{d}(x + x_i) = x^d + a_{d-1}x^{d-1} + \cdots + a_1 x + a_0$$

Let $f(x) = a_{d-1}x^{d-1} + \cdots + a_1 x + a_0$, Thus, $I(x) = x^d + f(x)$.

In order to prove that the degree of $I(x)$ is at least d it suffices to show that $\mathsf{deg}(f(x)) \leq d - 1$. This implies that there is no term of f that eliminates x^d. Therefore, the degree of polynomial I is at least d.

Proving that deg(f)$\leq$ $d - 1$. Proving a polynomial's maximum degree uses the technique from Marlin [51]. Instead of publishing parameters of degree-specific size, the accumulator's parameters of size t can be transformed and shifted to a polynomial such that it has degree t instead of d.

Informally, polynomial $f(x)$ gets multiplied with a random polynomial of degree $t - (d-1)$ (the sparse polynomial $c \cdot x^{t-(d-1)}$ where c is a random scalar would suffice). This is what we call a shift and can be proven to be computed correctly with the use of pairings.

Using knowledge assumptions [75] (and commitment to the same polynomial multiplied with a) the prover shows knowledge of f. In other words, the prover was able to construct the result polynomial using public parameters (that consist of generators raised to powers of s up to t), therefore the degree of f does not exceed $d-1$.

Protocol for $\mathcal{R}_{\mathsf{degcheck}}$. The interactive version of the protocol is as follows:

- Prover
  - Computes: $f(x)$ as $I(x) = \prod_{i=1}^{d}(x + x_i) = x^d + f(x)$, where $\mathsf{deg}(f) \leq d-1$
  - Sends $C_f = \frac{C_I}{g_2^{s^d}}$
- Verifier
  - Sends: $c \in \mathbb{Z}_p$
- Prover
  - Computes: $f(x) \cdot cx^{t-d+1}$ and $r \cdot cx^{t-d+1}$
  - Sends: $C = g_2^{f(s)cs^{t-d+1}} h_2^{r \cdot cs^{t-d+1}} \in \mathbb{G}_2$, $C^a \in \mathbb{G}_2$
- Verifier checks:
  - $e(g_1, C_I) = e(g_1, C_f) \cdot e(g_1, h^{s^d})$
  - $e(g_1, C) = e(g_1^{cs^{t-d+1}}, C_f)$, $e(g_1, C^a) = e(g_1^a, C)$

Asymptotics. The protocol has $O(d)$ prover cost that comes from multiplying $f(x)$ with the polynomial of degree $t-(d-1)$ and computing its commitment C. It can be made non-interactive using the Fiat-Shamir transformation. The proof size is constant (the prover has to send over constant sized commitments C_I, C_f, C, C^a: a commitment to polynomial $I(x)$, to polynomial $f(x)$, to polynomial $f(x) \cdot c \cdot x^{t-(d-1)}$ and to polynomial $a \cdot f(x) \cdot c \cdot x^{t-(d-1)}$ respectively) and verification cost is also constant (7 pairings).

4.8 Evaluation

In this section, we experimentally compare the aggregation operations in the RSA [26] and BP setting. We implement RSA accumulator using C++17, GNU Multiple Precision Arithmetic library 6.2.1 [152], and OpenSSL 3.0.2 [135]. We choose two 1024-bits prime numbers at random and compute the product to obtain a 2048 RSA modulus (using OpenSSL [135]). We implement [2] the BP accumulator using Golang bindings of the `mcl` library [99, 140]. Specifically, we use BLS12-381, a pairing-friendly elliptic curve. A single group element $\mathbb{G}$ and a field element $\mathbb{F}$ in the RSA setting, by the virtue of the choice of parameters, are 256 and 32 Bytes, respectively. In the elliptic curve group, a single compressed $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ group element requires 48, 96, 576 Bytes, respectively. Moreover, an element in $\mathbb{Z}_p$ requires 32 Bytes. A single exponentiation in the RSA group $\mathbb{G}$ by an exponent at most 256-bits takes 449 μs on an average. However, a single exponentiation in the elliptic curve source groups take $\mathbb{G}_1$ and $\mathbb{G}_2$ takes 106 μs and 250 μs, respectively. As BLS12-381 curve contains numerous roots of unity, we implement FFT based polynomial algorithms to support fast polynomial operations in `go-mcl` [140].

Our implementation is single threaded and all our experiments where performed on an Intel Core i7-4770 CPU @ 3.40GHz and 32 GiB of RAM. Unless stated otherwise, we perform 3 runs of each experiment and report the average.

[2]Our code is available at: https://github.com/accumulators-agg/accumulators

Operation	Sch.	2^9	2^{11}	2^{13}	2^{15}	2^{17}
		Batch size				
Domain mapping (s)	RSA	0.33	1.31	5.25	20.99	83.95
	BP	0	0	0	0	0
Commit (s)	RSA	0.52	2.09	8.38	33.55	134.37
	BP	0.05	0.24	1.12	5.17	24.28
AggMem (min)	RSA	0.04	0.17	0.8	3.66	16.49
	BP	0	0.01	0.18	2.51	38.35
MemVerify (s)	RSA	0.52	2.1	8.38	33.54	134.37
	BP	0.11	0.46	2.0	8.65	38.14
AggMemPoE (min)	RSA	0.04	0.19	0.86	3.87	17.33
	BP	0	0.02	0.2	2.58	39.12
MemVerifyPoE (s)	RSA	0.33	1.32	5.32	21.4	86.16
	BP	0.03	0.16	0.77	3.8	18.62
AggNonMem (min)	RSA	0.05	0.25	1.16	5.3	23.9
	BP	0.1	1.54	24.54	N/A	N/A
NonMemVerify (s)	RSA	0.72	2.87	11.49	45.98	184.12
	BP	0.11	0.46	2.0	8.65	38.14
AggNonMemPoE (min)	RSA	0.07	0.3	1.37	6.14	27.24
	BP	0.1	1.55	24.58	N/A	N/A
NonMemVerifyPoE (s)	RSA	0.34	1.34	5.33	21.41	86.17
	BP	0.03	0.16	0.77	3.8	18.62

Table 4.2: Accumulator batching operation costs for different batch sizes. In the first column, (s) denotes seconds and (min) minutes. The costs for RSA operations include the computations required to map to the prime domain. N/A stands for very large costs which are not interesting to compute.

4.8.1 Aggregation

In Table 4.1, we present the asymptotic costs for various operations in the BP setting, and in Table 4.2 we present our corresponding evaluation results.

Public parameters. Both the RSA and the BP accumulators require a trusted setup phase to generate the public parameters. Classgroups based accumulator constructions [31] do not require trusted setup in the unknown-order group setting, but they are too slow in

practice. The public parameters in the RSA setting is just the RSA modulus and the group generator. However, in the BP setting, the public parameter consists of $n \cdot \mathbb{G}_1 + n \cdot \mathbb{G}_2$ elements, where n is the maximum size of the accumulated set[3]. For $n = 2^{17}$, the public parameters occupies 18 MiB. With PoE, it is sufficient to store *only* constant number of values (g_1, g_1^s, g_2, g_2^s) from the public parameter by the verifier. Recall that g_1, g_2 are the group generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

Domain mapping. To add values from an arbitrary domain $\mathcal{D}$ to the accumulator set, each element has to be mapped to a value in the specific accumulator domain $\mathcal{D}'$. Since the RSA accumulator requires a prime domain, we implement the standard "hash to prime" algorithm [22, 26], where the hash operation is successively applied on the input until the hash function returns a prime value. We use Blake2s hash implementation in OpenSSL [135] to convert a value from arbitrary domain to Prime domain and GMP's Miller-Rabin's primality testing (15 rounds). For the BP setting, as discussed in Section 4.2.3, the specific domain is $\mathbb{Z}_p$. In our implementation we use the BLS12-381 curve for which the group order is around 256-bits [27]. Thus, we can accumulate any arbitrary string of size up to 256-bits without the need of any mapping.

In our experiments, we consider the accumulation of arbitrary 256-bit strings. In Table 4.2, we report the cost mapping these arbitrary strings to the accumulator domain. Mapping an arbitrary string to the prime domain takes 640 to 894 μs for 15 to 50 rounds of Miller-Rabin primality testing. Before performing an accumulator operation, all elements

[3]Rather than relying on a trusted entity, it is possible to use an MPC based setup ceremony to generate public parameters. We can adopt approaches from real-world MPC ceremonies of Zcash and AZTEC protocol which have successfully generated SNARK parameters for circuit sizes 2^{21} and 2^{27}, respectively [30] [147].

have to be converted to the accumulator domain. Thus, we include the cost of mapping to the accumulator domain for all operations in Table 4.1.

Discussion. We now briefly discuss what could go wrong in a universal accumulator if we don't include the mapping process during verification.

Consider the two most popular $\mathsf{H}_{\mathsf{prime}}$ approaches [22, 26][4] for mapping a non-prime element x to a prime y: (1) perform repetitive hashing $H(H(\dots H(x))) = y$ until a primality test indicates that the output is a prime, then outputs x and r where r is the number of hashing rounds required, (2) perform repetitive hashing of the value $(x||r)$, where r is a random nonce, until $H(x||r) = y$ is a prime. In both cases, when a prover wishes to add element x to the RSA accumulator, it first calculates the mapping y and stores r. The element y is accumulated. When they wish to prove (non-)membership, they provide r as a witness along with the proof to decrease verification costs. However, if a malicious prover can find two numbers of hashing rounds r_1, r_2 for the same element x, that correspond to two elements y_1, y_2 in $\mathcal{D}'$, then if say y_1 was the accumulated value (for x), the malicious prover could use y_2 to argue non-membership for x.

In order to avoid such attacks in universal accumulators[5], before verifying any (non-)membership proof, *all* verifiers need to check that the given prime mapping y is the first one that corresponds to the arbitrary element x and thus need to run all the repetitive steps the prover does. If the proof is a batch proof for $|I|$ elements, the verifiers need to repeat the process $|I|$ times individually for each element. The same holds for updates

[4] A recent work [105], attempts to optimize the "hash to prime" approach, by using Pocklington primality certificates in order to reduce the cost of primality testing on the side of the verifier. However, it still does not guarantee a deterministic mapping.

[5] If the accumulator does not support non-membership, then this attack does not apply.

(addition or deletions) to ensure primality since they might be initiated by an untrusted entity.

Commit. To commit to a set I in the RSA setting, we first compute the product of the elements in the set. Then, we perform modular exponentiation of this large product of size $O(\lambda \cdot |I|)$-bits. However, for BP accumulators, we first compute the coefficient of the accumulator polynomial using the subproduct algorithm and fast polynomial multiplication. Then, we perform a single multi-exponentiation of size $|I|$. Observe that from Table 4.2, even after subtracting the domain mapping costs from the Commit, BP accumulators are faster! For a set size of 2^{15}, it takes around 12.56 seconds to perform Commit, in the RSA setting, whereas it takes just 5.17 seconds in the BP setting.

Membership aggregation. We implement the membership proof aggregation algorithm from Boneh et al. to aggregate a set of membership proofs in RSA accumulators [26]. Aggregating a pair of membership proofs involves computing Shamir's trick, which requires computing the Bézout coefficients and performing two exponentiations. However, aggregating membership proofs in BP accumulators, involves $O(|I| \log^2 |I|)$ field operations and one $|I|$-sized multi-exponentiation (Table 4.1). Thus, we observe that the prover's cost to aggregate is lower for the BP accumulator for set sizes up to $|I| = 2^{15}$. Beyond these set sizes, the field operations in BP dominates aggregation cost. It is not very common for a prover to hold (or wish to aggregate) more than 2^{15} proofs. Thus, for most applications BP should be preferable.

We also implement PoE from Boneh et al. [26] and from Section 4.6 to optimize the verification of the aggregated proof in RSA and BP accumulator, respectively. Since the

Operation	**RSA** (Bytes)	**BP** (Bytes)
Digest	256	48
Mem. proof	256	48
Non-mem. proof	288	80
Agg. mem. proof (Naive)	256	48
Agg. non-mem. proof (Naive)	3.85* MiB	144* Bytes
Agg. mem. proof (PoE)	512	192
Agg. non-mem. proof (PoE)	1312	336
PoE	256	144 or 96
PoKE	544	×

Table 4.3: Sizes of accumulator digest and proofs in bytes. Asterisk(*) denotes a batch size of 2^{17}.

prover overhead in computing the aggregated proof is dominated by field operations in both RSA and BP accumulators, the additional exponentiations overhead incurred by a PoE enabled prover is limited. We observe this in our experiments as the prover engaging PoE additional incurs only around 12.7 and 4.2 seconds for RSA and BP accumulators, respectively, for 2^{15} values (Table 4.2).

We observe that verifying batch proofs (without PoE) in BP accumulator is 3.5× to 4.7× faster than RSA accumulators. This is because the multi-exponentiations in elliptic curve group $\mathbb{G}_1$ is faster than a single large exponentiation in the RSA group $\mathbb{G}$. In the PoE enabled setting, we observe that BP verification is 4.6× to 11× faster than RSA. In addition to the domain mapping costs, we also include the overhead to compute the Fiat-Shamir coins in our experiments.

Non-membership aggregation. The prover's cost to aggregate non-membership proofs is better for RSA regardless of the use of PoE. This is due to BP's $O(|I|^2 \log |I|)$ field operations that comes from constructing the $Y_i(s)$ terms (Section 4.4.2).

In comparison with RSA, verifying batch non-membership proofs in BP is at least $4.6\times$ faster in any case. Observe that verifying a batch non-membership is computationally similar to verifying a batch membership proof in the BP setting, regardless of the usage of PoE (Table 4.1). Thus, we observe similar performance numbers for NonMemVerify and NonMemVerifyPoE when compared to MemVerify and MemVerifyPoE, respectively, in the BP setting.

Storage and proof sizes. In Table 4.3, we present the storage overhead of proofs in both RSA and BP setting. For a similar level of security, an RSA group element $\mathbb{G}$ is of size 256 bytes as opposed to an elliptic curve element that is of size 48 Bytes for $\mathbb{G}_1$ and 96 Bytes for $\mathbb{G}_2$. The accumulator value and the batch membership proof consist of one group element in both constructions. Non-membership consists of one group element and one integer for RSA and two group elements in the BP setting. The integer in RSA batch non-membership proof grows linear in the batch size. The PoE proof adds to the proof size one group element in the RSA. However, in the BP setting PoE adds an overhead of either $(\mathbb{G}_1, \mathbb{G}_1)$ or $(\mathbb{G}_1, \mathbb{G}_2)$ depending on whether prover computes a proof for $g_2^{I(s)}$ or $g_1^{I(s)}$, respectively. The RSA non-membership proof can be made succinct using PoE and PoKE [26]. We note that in BP, non-membership proofs do not need PoKE as the proofs are already constant sized. Thus, we observe that membership and non-membership proofs

Operation	Batch size 2^9	2^{11}	2^{13}	2^{15}
Pedersen Commitment (s)	0.08	0.31	1.24	4.98
Prover $\mathcal{R}_{\mathsf{mem}}$	2.97 ms			
Verifier $\mathcal{R}_{\mathsf{mem}}$	3.66 ms			
Proof size $\mathcal{R}_{\mathsf{mem}}$	0.91 KiB			
Prover $\mathcal{R}_{\mathsf{nonmem}}$	4.65 ms			
Verifier $\mathcal{R}_{\mathsf{nonmem}}$	5.18 ms			
Proof size $\mathcal{R}_{\mathsf{nonmem}}$	1.03 KiB			
Prover $\mathcal{R}_{\mathsf{degcheck}}$ (s)	0.17	0.64	2.57	11.29
Verifier $\mathcal{R}_{\mathsf{degcheck}}$	4.49 ms			
Proof size $\mathcal{R}_{\mathsf{degcheck}}$	0.29 KiB			

Table 4.4: Single-threaded microbenchmarks for our ZK constructions.

in BP accumulators are 2.5× to 5× smaller and 3.5× smaller than the RSA accumulators, respectively.

4.8.2 Zero-knowledge batch proofs

We microbenchmark our proposed ZK batch proofs in Table 4.4.

Public parameters. The constructions for $\mathcal{R}_{\mathsf{mem}}$ and $\mathcal{R}_{\mathsf{nonmem}}$, require the prover and the verifier to store additional generators for Pedersen commitment. However for $\mathcal{R}_{\mathsf{degcheck}}$, since we rely on t-PKE assumption (Assumption 2.1.3), the prover needs to additionally store $3n \cdot \mathbb{G}_2$ elements (36 MiB). Whereas, the verifier needs to store only $2n \cdot \mathbb{G}$ elements (24 MiB). When $n = 2^{17}$, generating additional $3n \cdot \mathbb{G}_2$ parameter takes around 98 seconds using a single thread.

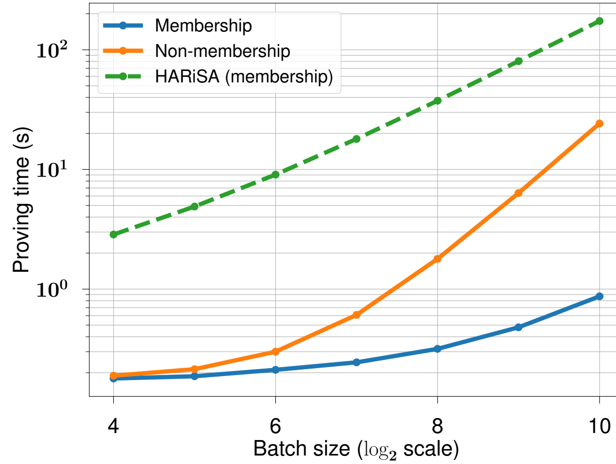


Figure 4.3: We extrapolate the proving costs using the numbers reported in HARiSA [43]. Note that the results in the RSA setting does include the Hash-to-prime costs.

Prover overhead. To commit to coefficients of a polynomial using Pedersen commitment, we use the values $g_2^{s^i}$ and an independent group generator. Using multi-exponentiation, it takes 4.98 seconds to commit to a batch of size 2^{15}. Recall that, given a commitment to the subset I, prover incurs constant overhead to generate a ZK proof for $\mathcal{R}_{\mathsf{mem}}$ and $\mathcal{R}_{\mathsf{nonmem}}$ regardless of the batch size. Thus, to prove $\mathcal{R}_{\mathsf{mem}}$ and $\mathcal{R}_{\mathsf{nonmem}}$, it takes 2.97 and 4.65 milliseconds, respectively. To prove a lower bound on the degree of $I(s)$, the prover needs to compute Pedersen commitments on $I(s)$ with and without t-PKE.

Verification time and proof size. The proof of $\mathcal{R}_{\mathsf{mem}}$ is $(4 \cdot \mathbb{G}_1 + \mathbb{G}_T + 5 \cdot \mathbb{Z}_p)$ and the proof of $\mathcal{R}_{\mathsf{nonmem}}$ is $(3 \cdot \mathbb{G}_1 + \mathbb{G}_2 + \mathbb{G}_T + 6 \cdot \mathbb{Z}_p)$. With a 64-bit integer to denote the degree and three elements in $\mathbb{G}_2$, a prover can prove a lower bound on the degree of a polynomial. All the proofs in our scheme can be verified with a constant number of exponentiations and pairing operations.

4.8.3 Comparison with HARiSA [43]

In this subsection, we argue that our approach to ZK batch proofs of membership can be at least 16× faster than the current state-of-the-art approaches to ZK batch membership proofs in the RSA setting for a reasonable choice of batch size. We also report the performance of our ZK batch proof of non-membership in Fig. 4.3. HARiSA does not support non-membership.

Experimental setup. We fix the maximum size of the set to 2^{17} elements and measure the performance of computing the zero-knowledge proof of batch (non-)membership while revealing the size of the batched subset. Moreover, we consider an experimental setup where the prover must do maximal work. That is, we assume that the prover: (1) has access only to the individual (non-)membership witness but not the batch membership witness, (2) does not have access to the commitment to the batched subset, and (3) has access to the accumulator digest and public parameters. Thus, the prover incurs the cost of: (1) computing the commitment to set I, (2) aggregating the individual (non-)membership witnesses to obtain batch witness, (3) proving the relation $\mathcal{R}_{\mathsf{mem}}$ or $\mathcal{R}_{\mathsf{nonmem}}$, and (4) proving the relation $\mathcal{R}_{\mathsf{degcheck}}$ for $d = |I|$.

Baseline measure. We use HARiSA by Campanelli et al. as the baseline measure to benchmark our scheme [43]. HARiSA builds a succinct batch proof of membership while preserving the privacy of the batched elements [43]. Their work combines proof of knowledge of exponent (PoKE) along with CP-SNARK for integer arithmetic relations and bound checks to prove batch membership. They implement their construction using LegoGroth16 in C++. Similar to our experiments, they require the prover to compute the batch witness

using individual witnesses and their experiments are single threaded. Also, recall that in our experiments we reveal the size of the subset I, which is currently not implemented in HARiSA [43].

Proving time. HARiSA reports a prover time of 2.86 and 9.02 seconds for a batch of size 16 and 64, respectively. Recall that their implementation uses LegoGroth16 proof system, thus the prover time is dominated by large FFTs and exponentiations. The performance numbers reported by HARiSA [43, Figure 4] uses Amazon EC2 r5.8xlarge. However, in Fig. 4.3, the performance of our scheme is measured on an Intel Core i7-4770. Unfortunately, we encountered challenges running the baseline code on our testbed despite our best efforts. The available documentation and support provided limited guidance in resolving these issues. To address this setback, we extrapolated the numbers using available data for comparison in Fig. 4.3. We observe that for a batch size of 16, our approach takes merely 0.18 seconds, whereas HARiSA takes 2.86 seconds, thus resulting in 16× speed up. Our performance is still an order of magnitude faster even when benchmarked on a commodity machine and even after additionally revealing the subset's size. Thus, we argue that our order of magnitude performance gain will likely carry over even when we benchmark our scheme and HARiSA on the same testbed.

Verification time and proof size. Our approach has a comparable proof size and better verification speed. While the verification performance of the baseline can be improved using multipairings, factoring that, we would like to highlight that our verification speed is expected to continue to remain better. We report the numbers as-is in Table 4.5.

Scheme		Setup	Verifier	Proof size
HARiSA [43]	Mem.	Trusted	63 ms	1.14 KiB
This work	Mem. Non-Mem.	Trusted	7.94 ms 9.41 ms	1.2 KiB 1.32 KiB

Table 4.5: Verification overhead and proof size.

4.9 Example: Non-membership aggregation

Example: $|I| = |\{y_1, y_2, y_3\}| = 3$. Let $I = \{y_1, y_2, y_3\}$ be a set of elements disjoint from the accumulated set X, $\overline{\pi}_1 = (\alpha_1, {g_1}^{\beta_1(s)})$, $\overline{\pi}_2 = (\alpha_2, {g_1}^{\beta_2(s)})$, and $\overline{\pi}_3 = (\alpha_3, {g_1}^{\beta_3(s)})$ be the non-membership proof of the element y_1, y_2, and y_3, respectively. Let, $X(s) = \prod_{x \in X}(s+x)$ be the accumulator polynomial, $I(s) = (s+y_1)(s+y_2)(s+y_3)$ be the accumulator polynomial of I, and $Y_i(s) = \frac{I(s)}{(s+y_i)}$. Thus, $Y_1(s) = (s + y_2)(s + y_3)$, $Y_2(s) = (s + y_1)(s + y_3)$, and $Y_3(s) = (s + y_1)(s + y_2)$. Note that $\gcd(Y_1(s), Y_2(s), Y_3(s)) = 1$. By generalization of Bézout's identity for polynomials, there exists polynomials $c_i(s)$, such that:

$$c_1(s) \cdot Y_1(s) + c_2(s) \cdot Y_2(s) + c_3(s) \cdot Y_3(s) = \gcd(Y_1(s), Y_2(s), Y_3(s)) = 1$$

To compute the non-membership proof of I, we need to compute polynomials $\alpha(s)$ and $\beta(s)$ such that:

$$\alpha(s) \cdot X(s) + \beta(s) \cdot \prod_{y_i \in I}(s + y_i) = \gcd(X(s), \prod_{y_i \in I}(s + y_i)) = 1$$

Thus, from Eq. (4.2) we have:

$$\alpha(s) = \alpha_1 c_1 Y_1(s) + \alpha_2 c_2 Y_2(s) + \alpha_1 c_3 Y_3(s)$$

$$\beta(s) = c_1 \beta_1(s) + c_2 \beta_2(s) + c_3 \beta_3(s)$$

Part II

Protocols to Improve Security in Distributed Consensus

Chapter 5: Permissionless Protocol in the Mobile Sluggish Model

Nakamoto's protocol, used in Bitcoin, achieves consensus over the Internet in a *permissionless* setting, where: any node can join and leave the system at any time, the exact number of participating nodes is unknown, and the nodes have to communicate over unauthenticated channels. However, for security, the protocol assumes that the network is synchronous – all honest messages get delivered to one another within a known upper bound on time, Δ units.

Unfortunately, assuming that an Internet scale protocol is synchronous is excessively optimistic. Moreover, Pass and Shi [114] showed that it is *impossible* to achieve permissionless consensus in an asynchronous or even in a partially synchronous network [14], which are relaxations of the synchronous model. Thus, to deploy the protocol in the real-world, the protocol designers are compelled to choose a loose upper bound Δ as the network delay to accommodate nodes with slow network.

Guo et al. introduced a relaxation of the synchronous model, which was subsequently called the *mobile sluggish model* [5, 77]. In the mobile sluggish model, a fraction of honest nodes, called *sluggish* nodes, can arbitrarily lose synchrony, but they faithfully follow the rest of the protocol. The remaining honest nodes, called *prompt* nodes, are synchronous and faithfully follow the protocol. Additionally, sluggishness can be *mobile*, that is, any honest

node can become sluggish over the protocol execution, and if a sluggish node becomes prompt by regaining synchrony, it will receive all the backlogged messages. This model is stronger than the partially synchronous and asynchronous model but weaker than the synchronous model.

In this work, we relax the standard synchrony assumption and study Nakamoto consensus under the mobile sluggish model [5, 77]. For Internet scale protocols, the sluggish model is a pragmatic trade-off between the synchronous model and partially synchronous/asynchronous model.

One way to defend against a mobile sluggish adversary is to let an honest block winner simply time-lock encrypt the message before sending it, and other honest nodes time-lock encrypt a *decoy* to distract the adversary from spotting the block winner. Since the adversary cannot learn the contents of the puzzle without spending sufficient time, by setting the TLP duration slightly greater than the round duration, the adversary is now forced to corrupt or deliver the message randomly. At the end of the round, honest nodes can batch solve the TLPs they received and update their chains. Unfortunately, *no* prior TLP with batch solving works in this application, due to the requirements and challenges in the permissionless setting: we cannot rely on a trusted setup [95, 122, 136], we do not know the number of users in the network a priori, and we do not want to blowup the round [122] and communication complexity [95, 136, 137].

Thus, we ask the following question:

Is it possible to achieve consensus in a permissionless setting

in the presence of mobile sluggish faults?

We affirmatively answer this question by proposing a protocol that uses our TLP construction from [132] as a fundamental building block to show that it is possible to achieve *consistency* (any two prompt chains can differ only in the last few blocks) and *liveness* (every prompt node eventually commits all transactions) even in the presence of mobile sluggish faults.

5.1 Technical overview

In this section, we show how to adapt the Nakamoto consensus to defend against a mobile sluggish adversary using the our TLP. In our protocol, we use the following ideas: (1) All honest nodes time-lock encrypt any message they transmit, (2) all honest nodes send *decoys* to protect the block winner from getting caught by the adversary, (3) restrict the adversary from flooding with decoys, and (4) ignore malformed puzzles sent by the adversary.

Formally, we define a round, a super-round, and duration of a round in Section 5.3 and Section 5.4. However, as a warm-up, we present strawman solutions to illustrate the inadequacies of the well-known approaches.

Strawman solutions. The first straightforward solution is to use RSW puzzles to time-lock encrypt any message with a duration equal to the network delay before transmitting across the network [122]. Unfortunately, this approach does not work for the following reasons:

- Recall that in a protocol like the Nakamoto consensus, only the block winner sends a message to the network. Thus, the adversary can easily stop the one message transmitted, whether or not the message is encrypted.

- Say the other honest nodes send out time-lock encrypted dummy messages, which act as a *decoys* to protect an honest block winner from getting caught. Unfortunately, the honest parties have to open all the puzzles to find the winning block. Thus, the honest parties either have to open all the puzzles individually or open them using the *distributed-solve* primitive proposed by Wan et al. [148, Section 4.2]. Both these approaches increase the round complexity of the protocol by linear and polylogarithmic rounds, respectively.

An alternate approach is to use TLPs with batch solving property defined in [136,137], but we would suffer from large communication costs and fixed batch size problem as explained before. Instead, we can now use our TLP that gets rid of the these issues. Below we give an overview of other challenges we encounter in designing our permissionless consensus protocol.

Decoys, spam prevention, and malformed puzzles. Since the Nakamoto consensus is in the permissionless setting, there are no identities to tackle Sybil attacks. This setting raises an important question: how to stop the adversary from spawning multiple identities to send decoys? We resort to proof-of-work to tackle the Sybil attack!

Say the difficulty threshold to mine a block is T, then we set the threshold to mine a decoy as T_c, such that $T < T_c$. Each RO query made by a node simultaneously tries to mine a block and a *decoy*. That is, say h is the output of the hash function. If $h < T$, then *a block* is mined, else if $T \leq h < T_c$, then *a decoy* is mined. This is the "2-for-1 POW" trick introduced by Garay et al. [12, 66, 113]. The parameter T_c presents an interesting

trade-off: T_c should be sufficiently high so that honest nodes mine enough decoys whereas the adversary should not be able to overwhelm the honest nodes with many decoy puzzles.

One of the challenges is that nodes do not know the exact number of decoys mined at a given time. However, since our TLP construction can batch a variable number of puzzles, nodes can flexibly batch puzzles on demand. Observe that T_c restricts the number of decoys that the adversary (and the honest nodes) can mine. But, this does not stop the adversary from flooding the honest nodes with malformed puzzles. Batching malformed puzzles along with honest puzzles prevents a node from obtaining the solutions to honest puzzles. To circumvent this problem we equip our TLP with a verifiability property that allows an honest node to reject a puzzle that is not *well-formed* according to the puzzle generation algorithm. Thus, a valid proof guarantees that the plaintext can be obtained by solving the puzzle.

Mine phase and solve phase. Since the mining process is stochastic, the arrival times of a decoy and a block are random. Say if an honest node sends the puzzle as soon as it finds the block, it is unlikely that the rest of the honest nodes will also be sending the decoy puzzles at the same time. If enough honest nodes do not provide "cover" to the block winner, then the probability of the adversary guessing the block winner is high. However, if all honest nodes wait until a pre-determined time to send the respective puzzles, then block winner will have the best chance of not being detected by the adversary.

In order to capture this intuition, we have two phases in our protocol:

- *Mine phase*: All nodes spend a sequence of m rounds mining a block or decoy without sending or receiving any messages.

- *Solve phase*: This phase begins as soon as the mine phase ends and consists of *two* rounds. In the first round, nodes send and receive the puzzles they have mined in the *mine phase*, and check the well-formedness of the received puzzles. In the second round, nodes will batch solve the TLPs to find the block, if any, and update the longest chain.

  We generically denote the duration of the solve phase as D rounds. If one employs RSW puzzles and the *distributed-solve* primitive from Wan et al. [148] instead of our TLP construction, then D can be thought of as the number of rounds required to perform *distributed-solve* procedure. However, when our protocol is instantiated with our TLP construction we have $D = 2$.

Thus, the duration of a super-round is $(m + D)$ rounds.

Putting it all together. In summary, by using our TLP, the decoy mechanism, and super-rounds, our protocol works as follows: Every honest node performs the following steps in every super-round: (1) Receive transactions from the environment, (2) choose the longest chain it has seen so far and break ties arbitrarily, (3) mine for m rounds (mine phase), and (4) solve for D rounds (solve phase) and update the longest chain. We defer the details of the protocol to Section 5.4.

5.2 Attack on Nakamoto consensus in the mobile sluggish model

In this section, we show an attack against the Nakamoto consensus in the mobile sluggish model and study the impact of mobile sluggish nodes on the resilience of the protocol.

In Nakamoto consensus, a chain forks when two distinct blocks extend the same parent block. Forks are inherently bad for security as it splits the honest mining efforts across the two branches of the tree. A benign example is when two blocks are mined less than Δ time units apart. Since the messages take Δ to reach others, the winner of the second block would not have been aware of the previous block. Nakamoto consensus is parameterized in a way that the inter-arrival time between two blocks is much longer than the time to transmit between any two farthest nodes in the system. The security threat posed by forks is the exact reason the Nakamoto consensus is secure only in the synchronous model.

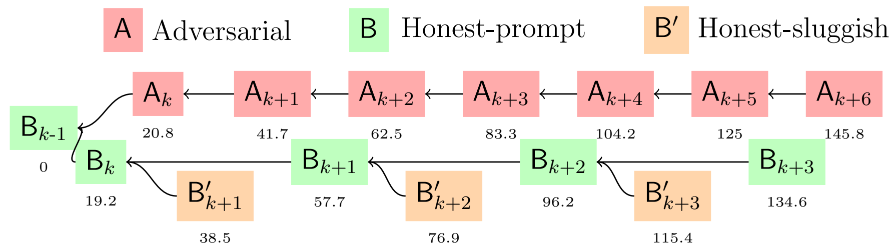


Figure 5.1: Double spend attack: This plot depicts average block arrival times. Assuming 52 honest nodes (51 prompt + 1 sluggish) and 48 adversarial nodes, the average inter-arrival time of honest blocks and adversarial blocks in Bitcoin is 19.2 and 20.8 minutes, respectively. Observe that over 19.2×2 minutes, even though the honest nodes have mined two blocks, due to sluggishness, the honest chain has grown only by one block.

The mobile sluggish adversary, whenever an honest node mines a block, can simply delay the block propagation until another block extends the same header (see Fig. 5.1). At this point, the adversary can release both the blocks simultaneously to split the honest mining efforts. The adversary can sustain the forks as long as it has sufficient sluggish budget. Since the adversary is responsible for message delivery and sluggishness can be mobile, it could perform this attack repeatedly. In the meantime, adversarial nodes will continue to extend their chain in private. Using this strategy, even a single mobile sluggish

fault has the ability to reduce the honest mining rate by *half*! Thus, an honest majority assumption may not be sufficient to guarantee security in this model. In the extended version of paper, we detail the attack illustrated in Fig. 5.1 [131].

5.3 Model

Let n be the total number of nodes, d be the maximum number of sluggish nodes, and t be the maximum number of adversarial nodes. Thus, there are at least $n - t$ honest nodes and at least $n - d - t$ prompt nodes. We adopt the formal framework from Garay et al. [66], a model inspired by the prior formulations of secure multiparty computation [44].

Sluggish network model. We assume that the time proceeds in rounds. Moreover, we assume that the adaptivity of the adversary is static. That is before the protocol execution, the adversary picks the set of nodes to corrupt. Moreover, we also assume that every node has access to a *shared global clock* and a pairwise reliable channel between any two parties.

The standard (lock-step) model of synchrony assumes that any message sent in round r reaches other nodes by $r+1$. We consider a generalization of this model called the *mobile sluggish* model. In this model, if a node is prompt at round r, then any message sent by the node in round $\leq r$ reaches all the nodes that are prompt in round $r+1$ by round $\leq r+1$. Due to mobility of the sluggishness, set of prompt nodes in any two adjacent rounds need not be the same.

The adversary is responsible for message delivery. Thus, an adversary can reorder or delay messages (according to prompt and sluggish delay requirement), but *cannot* delete messages. Moreover, any message sent to a prompt node *by a prompt or an adversarial node*

reaches all prompt nodes. We can relax this assumption by assuming that nodes gossip/echo any message they receive [112, Footnote 4]. The adversary inspects all messages (including puzzles and blocks) first before delivering to any node.

Round duration. We assume that the duration of a round is $O(\Delta)$. Specifically, we assume that a round is sufficiently long to send/receive messages and perform cryptographic operations (such as verifying a hash of a message, generating and verifying a zero-knowledge proof of well-formedness of a TLP, computing $\mathsf{PGen}/\mathsf{PEval}$, and signing and verifying a signature), *except* $\mathsf{PSol}, \mathsf{BatchPSol}$, and RO invocations to mine a block or a decoy.

Computational model. We adopt the *flat* model of computation introduced by Garay et al. [66]. In this model, all nodes are assumed to have the same computational power. Moreover, any node can make at most q proofs-of-work invocations to the RO in a round. Thus, the adversary can perform $t \cdot q$ RO queries in each round. We remark that each node has an unlimited number of proof-of-work verification queries to the RO [66].

Additionally, we assume that all honest nodes are sequential, random access PPT, but the adversary is a non-uniform probabilistic parallel machine with polynomially bounded parallelism running in polynomially bounded parallel steps.

Environment. The entity *environment* handles the external aspects of the protocol execution such as spawning the nodes and the adversary, injecting transactions, writing inputs and reading outputs of each node, etc. However, the environment *cannot* make queries to RO. This is to prevent the adversary from outsourcing the RO queries to an external entity.

5.4 Protocol

In this section we present a permissionless protocol to defend against mobile sluggish adversaries using the TLP from [132].

Super-round. Since our protocol proceeds in two phases: (1) Mine phase (m rounds) and (2) Solve phase (D rounds), a super-round consists of a mine phase followed by a solve phase. Thus, the duration is $(m + D)$ rounds.

Mobile Sluggish Nakamoto Protocol

Input.

- pp, TLP public parameters with $\mathbf{T}$ as one round
- m, duration of mine phase
- D, duration of solve phase
- q, maximum number of RO queries per round
- T, difficulty threshold to mine a block
- T_c, difficulty threshold to mine a decoy where $T < T_c$

Initialize. Chain C containing agreed-upon genesis block $\mathsf{C}[0]$

Protocol. Every super-round R (which consists of $(m + D)$ rounds)

- Get the $\mathsf{payload}$ from the environment
- Let $h_{-1} := \mathsf{H}(\mathsf{C}[-1])$ be the hash of the last block on the longest chain C
- Let $B = \perp$ be an empty block
- For m rounds of **mine phase**:
  - For q RO queries:
    - Pick random $\eta \in \{0,1\}^\lambda$ and compute $h := \mathsf{H}(h_{-1}, \mathsf{payload}, \eta)$
    - If $h < T_c$ (mined a decoy)
      - Overwrite $B := (h_{-1}, \mathsf{payload}, \eta)$
    - If $h < T$ (mined a block)

· Overwrite $B := (h_{-1}, \mathsf{payload}, \eta)$
· Set $\mathsf{C} := \mathsf{C} || B$
· Break out of the q and m loop

- If $B \neq \bot$
  - Compute the TLP $Z := \mathsf{PGen}(\mathsf{pp}, B)$
  - Compute proof of well-formed $\pi := \mathsf{PProve}(\mathsf{pp}, Z, B)$
- **Solve phase** for $D = 2$ rounds:
  - First round, multicast (Z, π) (if one exists), receive all the w puzzles from the network $Z_1, \ldots, Z_w$, and check their well-formedness.
  - Second round, batch solve $(s_1, \ldots, s_w) := \mathsf{BatchPSol}(Z_1, \ldots, Z_w)$.
- Update the chain C based on output from the solve phase

5.5 Analysis

Assumptions. Let a block mined in a super-round R be a *prompt block*, if mined by an honest node and the node was prompt *at the beginning* of solving phase of *both* $R-1$ and R. Moreover, let f be the probability of one or more prompt blocks were mined in a super-round, c be the probability of every honest node mining at least one decoy in a super-round, $\varepsilon, \delta \in (0,1)$ be parameters, and p be the probability of a RO query mining a block. Our analysis assumes that:

$$\frac{(m+D)t + md}{cm(n-2d-t)} \leq (1-\delta) \qquad (5.1)$$

$$\varepsilon + f < \delta/3 \qquad (5.2)$$

$$pqm(n-2d-t) < 1/2 \qquad (5.3)$$

$$\frac{2\varepsilon}{1-\varepsilon} < \delta^2 \qquad (5.4)$$

Analysis. At a high level, our analysis extends the formal tools proposed by Garay et al. [66]. But there are several differences due to mobile sluggish faults and the use of TLPs:

1. The adversary can deviate from the protocol and invoke RO queries even during the solve phase. Intuitively, Eq. (5.1) quantifies the required advantage of the prompt nodes over sluggish and adversarial nodes for our protocol to be secure. Specifically, the numerator captures the computational advantage enjoyed by the adversarial nodes due to additional RO queries during the solve phase (the term $(m + D)t$) and the loss in honest mining efforts due to sluggish nodes (the term md). Large values of D decreases t (assuming other values can remain the same). But, due to the batch solving property of our TLP, $D = 2$ in our protocol. Thus, the impact of D is minimal.

2. The mobility of the sluggishness provides the adversary timing based opportunities to reduce the contributions to the "prompt" chain. The adversary with d sluggish budget can toggle the sluggishness of $2d$ nodes. If the adversary toggles the sluggishness when the honest nodes release TLPs at the end of the mining phase, it can reduce the number of nodes contributing to the prompt chain to $(n - 2d - t)$. This is because the d nodes that are sluggish through the mining phase of a super-round may not be mining on the longest chain, and at the end of the mining phase, the adversary can use its mobility to make d prompt node sluggish (See [131, Remark 1]).

3. The sluggish nodes can inadvertently contribute to the adversarial chain. This is because the sluggish nodes may only have access to the view provided to them by the adversary.

4. Coordinated release of TLPs: Observe that from Eq. (5.3), large values of m decreases p, thus reducing the block arrival frequency. But, a bounded p ensures that the honest nodes do not fork one another and there are sufficient "convergence opportunities" to

resolve forks [66,112]. Moreover, no prior permissionless protocol is secure under mobile-sluggish faults even under reduced performance.

5. Impact of decoys: In Eq. (5.1), the security impact of mining decoys by honest nodes is captured by c. We set the probability of mining a decoy such that honest nodes can mine sufficiently many decoys while simultaneously bounding the total number of decoys mined. Recall that our batch solvable TLP allows simultaneously opening a polynomial number of puzzles.

Notice that our analysis is a generalization of [66], thus by substituting $m = 1, c = 1$, $d = 0$, and $D = 0$, our analysis, in principle, collapses to [66]'s analysis. We prove liveness and consistency by assuming that the mining-hardness parameter is appropriately set in Eqs. (5.1) to (5.4).

We present the complete analysis of the protocol in the extended version our paper [131].

5.6 Related work

Protocols in the mobile sluggish model. Guo et al. [77] first introduced the mobile sluggish model as "weakly synchronous" model and showed that it is impossible for a Byzantine broadcast protocol to tolerate majority faults (Byzantine or sluggish). Subsequently, Abraham et al. presented a Byzantine Fault Tolerant blockchain protocol that can tolerate minority corruptions in the mobile sluggish model [5]. Kim et al. [86] observed that many proof-of-stake protocols, such as Dfinity [78], Streamlet [48], OptSync [126], can support mobile sluggish faults. These prior techniques heavily relied on using messages

(votes) from a majority of the nodes (certificates) to establish communication with sluggish nodes and ensure safety of the protocol. Since Nakamoto consensus does not rely on such certificates, their techniques do not apply in our setting.

Nakamoto style protocols. Prior works can be categorized based on the flavor of synchrony used to analyze Nakamoto consensus. In the lock-step model of synchrony, Garay et al. formally analyzed Nakamoto consensus [66]. Subsequently, Pass et al. and Kiffer et al. showed that the Nakamoto consensus is secure even in the non-lock-step synchrony model where the message delay is bounded and the time proceeds in discrete rounds [66, 85, 112, 165]. Ren discarded the notion of discrete rounds and proved the security of Nakamoto consensus in the continuous model [120]. Even parallelly composed Nakamoto protocols are also in the lock-step model of synchrony [12, 157]. Unfortunately, all these analyses assume that *any* honest message reaches other honest nodes in Δ time units regardless of the flavor of synchrony. Our analysis is in the mobile sluggish model, which assumes that a fraction of honest nodes can violate the Δ-assumption. However, the prompt nodes in our setting are assumed to be in lock-step synchrony model.

Network-adversary lower bounds and impossibilities. Abraham et al. showed that a sub-quadratic protocol could not be resilient against a strongly adaptive adversary that can perform *after-the-fact* removal [4]. In Nakamoto consensus, delaying an honest block has the same effects as deleting the block. For example, if a newly mined block is delayed for a sufficiently long time, it could end up as an orphan block, which eventually gets pruned after the main chain stabilizes. Moreover, sluggishness can be mobile, thus making the sluggish adversary more powerful than the strongly adaptive adversary. Pass and Shi showed that it

is impossible to achieve permissionless consensus in the partially synchronous/asynchronous network [114]. In these network models, the adversary can *arbitrarily* partition the honest nodes. However, in our setting, the adversary can create only *minority* partitions. Thus, this impossibility does not apply.

Chapter 6: Byzantine Broadcast under Strongly Adaptive Adversaries

Byzantine broadcast (BB) is a classical problem in distributed consensus, where a *designated sender* holds a bit b and wants to transmit b to all n nodes in the presence of t faults. A BB protocol is secure if it can guarantee *consistency* (all honest nodes output the same bit) and *validity* (if the designated sender is honest, all honest nodes output the designated sender's input b).

In recent times, BB has emerged as a fundamental building block in blockchains [70]. We study the round-efficiency of BB under the dishonest majority setting. Prior BB protocols in the dishonest majority setting can broadly tolerate: (1) weakly adaptive or (2) strongly adaptive adversary. Both strongly and weakly adaptive adversary can corrupt honest nodes *on the fly*. But, a weakly adaptive adversary cannot perform *after-the-fact* removal.

Despite decades of study, the state-of-art round-efficient BB in the dishonest majority is in the weakly adaptive setting [149]. Thus, it raises the question:

Is it possible to achieve an expected constant-round Byzantine broadcast under strongly adaptive and corrupt majority?

We affirmatively answer this using PKI, RO, and *any* batch solvable TLP construction. Our solution is a *generic round preserving compiler* that can convert any weakly adaptive

BB protocol into a strongly adaptive one. Thus, our compiler can be ***efficiently realized*** using the batch solvable TLP constructions based on RSA or Class-groups [136, 137]. Our complier is additionally *communication* preserving when the batch solvable TLP is *compact*, individual puzzle size is independent of the number of puzzles batched [131].

6.1 Technical overview

The key ingredient in our compiler is that we use TLPs to hide the contents of the messages sent by the underlying protocol so that the strongly adaptive adversary cannot learn the contents of any message before honest nodes receives it. Prior works use the RSW puzzles to defend against a strongly adaptive adversary [53,122,148]. But, due to the inability to batch solve RSW proofs, opening all puzzles collectively adds an overhead of polylogarithmic rounds to any protocol [148]. Thus, we use batch solvable TLPs defined in [131, 136, 137] to remove the polylogarithmic communication overhead incurred by RSW puzzles [122].

Even though TLPs can prevent the adversary from inspecting the contents of the message, the primary challenge is in proving that the compiled protocol is secure against an adversary that can perform *after-the-fact* removal. This is because TLPs, apart from hiding the message contents for $\mathbf{T}$ time units, also serve as a commitment to the message inside the puzzle, which prevents the simulator from simulating the honest nodes without knowing the actual contents of the puzzle! Cohen et al. encountered a similar problem in the context of *fair* BB and proposed a *non-committing TLP* to overcome this challenge [53].

Non-committing TLPs. Informally, it allows the simulator to equivocate a TLP. That is, the simulator first generates and sends a "fake" TLP to the network, which can be later

"opened" be to *any* message. Thus, when the simulator is asked to explain the contents, it programs the RO to open the desired message.

Compiler overview. Abstractly, in a weakly adaptive protocol $\Pi_{\mathsf{bb-wa}}$, a node performs three *basic steps*: In every round, (1) receives the messages sent by other nodes, (2) performs the state transition based on the messages received and computes the messages to send, and (3) sends the messages to other nodes. Our compiler interleaves each step of $\Pi_{\mathsf{bb-wa}}$ with TLP operations to obtain a strongly adaptive protocol, $\Pi_{\mathsf{bb-sa}}$.

In a bit more detail, before sending a message in the compiled $\Pi_{\mathsf{bb-sa}}$, a node uses non-committing TLPs to encrypt the message it wants to send and computes the puzzle with proof of well-formedness of the puzzle. Thus, instead of sending the plaintext in $\Pi_{\mathsf{bb-wa}}$, a node in $\Pi_{\mathsf{bb-sa}}$ sends the puzzle, ciphertext, and the proof of well-formedness to other nodes. When a node receives puzzles, ciphertexts, proofs of well-formedness from the network, instead of opening one puzzle at time, $\Pi_{\mathsf{bb-sa}}$ uses the batchable TLP proposed in this work to obtain all the solutions simultaneously without incurring additional round complexity or communication complexity to open all the puzzles. Thus, after opening all the puzzles, a node in $\Pi_{\mathsf{bb-sa}}$ invokes the state transition function just like a node in $\Pi_{\mathsf{bb-wa}}$. This process is repeated for every round. We defer the details to Section 6.3.

6.2 Model and definitions

In our setting, there are n nodes, numbered 1 to n, running a distributed protocol where the identity of each node is known to one another through a PKI.

Communication model. We assume that each node has access to a shared global clock and all parties are connected by a pairwise reliable channel. We consider the standard synchronous model of communication where there is a known upper bound on the message delay (Δ). The protocols are executed in a round-based fashion, where the duration of each round is Δ time units. Any message sent by an honest node in a round reaches all other honest nodes by the beginning of the next round. Also, each node has access to the functionalities: $\mathsf{RECEIVE}$ and SEND. When a node u invokes $\mathsf{SEND}(m, \mathsf{recipients})$ in round $r-1$, then m is delivered to $\mathsf{recipients}$ using the pairwise reliable channels from u by round r. When a node u invokes $\mathsf{RECEIVE}$ in round r, then all messages that were sent to u using the pairwise reliable channels by round $r-1$ are returned. The adversary can read, rearrange, insert, and drop messages between any two nodes (if strongly adaptive). But, cannot forge signatures. Moreover, we also assume that each round is sufficiently long to perform standard cryptographic operations except $\mathsf{BatchPSol}$ and PSol.

Let $\mathcal{P}$ be the set of possible internal states of a node and $\mathcal{M}$ be the set of possible messages that can be sent and received by a node.

Definition 6.2.1 (Δ-secure Synchronous protocol)**.** *Let $\mathcal{F}_n$ denote the family of transition functions such that:*

$$\mathcal{F}_n = \{f_{r,u} : \mathcal{P} \times \mathcal{M}^n \to \mathcal{P} \times \mathcal{M}^n : u \in [n], r \in \mathbb{Z}\}$$

A synchronous *protocol Π_{sync} is executed by n nodes and proceeds in rounds. In every round r, every node $u \in [n]$, reads the messages addressed to it using the* $\mathsf{RECEIVE}$

functionality, updates its state and computes the messages to be sent using $f_{r,u}$, and sends the messages to intended recipients using the SEND *functionality.*

Protocol $\Pi_{\mathsf{sync}}(\lambda, \Delta)$

Setup.

- Let $S_{0,u}$ be the initial state of node $u \in [n]$
- Generate and publish public parameters

Protocol. A node $u \in [n]$, for each round r:

- Fetch messages from each sender: $m := (m_1, \ldots, m_n) \leftarrow \mathsf{RECEIVE}()$
- Compute next state and messages: $(S_{r+1,u}, m' := (m'_1, \ldots, m'_n)) \leftarrow f_{r,u}(S_{r,u}, m)$
- Send messages: $\mathsf{SEND}(m', \mathsf{recipients})$

Adversary model. The adversary can make at most t out of n nodes to arbitrarily deviate from the protocol execution, where $t < n$. Moreover, we assume that the adversary controls the delivery of all the messages in the network.

- We consider a *strongly adaptive adversary* that can corrupt nodes on the fly and perform *after-the-fact* removal.
- Whereas, a *weakly adaptive adversary* can *only* corrupt nodes on the fly, but cannot prevent the delivery of any message that was already sent.

Additionally, we consider a *rushing* adversary that can inspect the messages sent by any honest node before delivering it to other nodes. Moreover, we assume that honest nodes can irrecoverably erase (part of) its state and memory at any time.

Computational model. All honest nodes are sequential, random access PPT, but the adversary is a non-uniform probabilistic parallel machine with polynomially bounded parallelism running in polynomially bounded parallel steps.

6.3 Protocol

For an n node protocol Π, we define a deterministic function called *output derivation function* for each node $u \in [n]$. This function allows a node to compute its output bit for Π based on the transcript of public messages exchanged by the participants and public parameters.

Definition 6.3.1 (Output derivation function)**.** *Let* Π *be an* n *node protocol and* $\mathcal{Y}$ *denote the* ***public*** *transcript space of the protocol* Π*, then* $\mathcal{G}_n$ *denote the family of* ***output derivation*** *functions such that:*

$$\mathcal{G}_n = \{g_u : \mathcal{Y} \to \{0,1\} : u \in [n]\}$$

Functions in $\mathcal{G}_n$*, despite being deterministic, may not be efficiently computable without a party's keys.*

We recall the definition of a secure Byzantine broadcast protocol below.

Definition 6.3.2 ((Δ, t)-secure Byzantine broadcast)**.** *Let* λ *be the security parameter,* Δ *be the known upper-bound on the network delay, and node* $d \in [n]$ *be the* designated sender*. A protocol* Π *executed by* n *nodes with specified family of functions* $\mathcal{G}_n$*, where the* designated sender *holds an input bit* $b \in \{0,1\}$*, is a* (Δ, t)*-secure broadcast protocol tolerating at most* t *corruptions if it satisfies the following properties with probability* $1 - \mathsf{negl}(\lambda)$*:*

- *Consistency: If two honest nodes output bit* b_i *and* b_j *respectively, then* $b_i = b_j$*.*

- *Validity: If the designated sender is honest, then every honest node outputs the designated sender's input bit b.*

- *Termination: Every honest node u outputs a bit from $g_u(\mathsf{transcript})$, where $\mathsf{transcript}$ is the transcript from running Π.*

If the protocol can tolerate corruptions by a strongly adaptive and a weakly adaptive adversary, then it is ***strongly adaptive*** *(Δ, t)-secure and* ***weakly adaptive*** *(Δ, t)-secure, respectively.*

Let $\Pi_{\mathsf{bb-wa}}$ be a weakly adaptive protocol, we formally describe $\Pi_{\mathsf{bb-sa}}$ below:

Protocol $\Pi_{\mathsf{bb-sa}}(\lambda, \Delta, \Pi_{\mathsf{bb-wa}}, \mathcal{G}_n)$

Text in gray indicates the instructions from $\Pi_{\mathsf{bb-wa}}$.

Setup.

- Let $S_{0,u}$ be the initial state of node $u \in [n]$
- For each round r, $\mathsf{pp} \leftarrow \mathsf{PSetup}(1^\lambda, \Delta)$
- Generate and publish public parameters

Input.

- Let $b \in \{0, 1\}$
- If designated sender, d, then $S_{0,d} := S_{0,d} \cup b$

Protocol. A node $u \in [n]$, for each round r:

- Fetch messages from each sender: $m := (m_1, \ldots, m_n) \leftarrow \mathsf{RECEIVE}()$
- Parse message m_v as puzzle Z_v, ciphertext C_v, proof of well-formed π_v for all $v \in [n]$
- Check π_v's to verify if Z_v's are well-formed by $\mathsf{PVer}(\mathsf{pp}, Z_v, \pi_v)$
- Extract the individual solutions $(s_1, \ldots, s_n) \leftarrow \mathsf{BatchPSol}(\mathsf{pp}, Z_1, \ldots, Z_n)$
- Decrypt C_v's, set $m_v := C_v \oplus H(s_v)$ for all $v \in [n]$, and $m := (m_1, \ldots, m_n)$
- Set internal state for round r as $S_{r,u} := m_u$

- Compute next state and messages: $(S_{r+1,u}, m' := (m'_1, \ldots, m'_n)) \leftarrow f_{r,u}(S_{r,u}, m)$
- Pick $s \in \mathcal{S}$, $Z \leftarrow \mathsf{PGen}(\mathsf{pp}, s)$, and compute π to prove that Z is well-formed.
- Reassign $m'_u := (Z, S_{r+1,u} \oplus H(s), \pi)$ and $m'_v := (Z, m'_v \oplus H(s), \pi)$ for all $v \in [n] \setminus \{u\}$
- Set output messages as $m' := (m'_1, \ldots, m'_n)$ and erase $S_{r+1,u}$, s, π
- Send messages: $\mathsf{SEND}(m', \mathsf{recipients})$

Output.

- Let $\mathsf{transcript}$ be the public transcript of the protocol execution
- Return $b \leftarrow g_u(\mathsf{transcript})$

Expected constant-round Byzantine broadcast. Wan et al. [149] proposed an expected constant round BB protocol under a weakly adaptive and dishonest majority setting. Thus, using the compiler (Section 6.3), we can obtain resilience in the strongly adaptive setting!

6.4 Analysis

In this section, we formally prove the security of our complier presented in Section 6.3.

Theorem 6.4.1. *Let $\Pi_{\mathsf{bb-wa}}$ be a weakly adaptive (Δ, t)-secure Byzantine broadcast protocol with output derivation functions $\mathcal{G}_n$ and $\Pi_{\mathsf{bb-sa}}$ be the compiled strongly adaptive (δ, t)-secure protocol with output derivation functions $\mathcal{G}_n$, such that $\Delta = 2\delta$. If an $\mathcal{A}$ violates $\Pi_{\mathsf{bb-sa}}$ with probability at least p, then there exists an adversary $\mathcal{B}$ that violates $\Pi_{\mathsf{bb-wa}}$ with probability at least p.*

Overview. Suppose $\exists$ an $\mathcal{A}$ that can break $\Pi_{\mathsf{bb-sa}}$, then we build another adversary $\mathcal{B}$ that breaks $\Pi_{\mathsf{bb-wa}}$. At a high level, we show that every attack by $\mathcal{A}$ on $\Pi_{\mathsf{bb-sa}}$ can be

translated to an attack on $\Pi_{\mathsf{bb-wa}}$. Observe that $\mathcal{B}$ is as powerful as $\mathcal{A}$, except $\mathcal{B}$ cannot perform *after-the-fact* removal. Thus, to translate the *after-the-fact* removal, $\mathcal{B}$ must know whether $\mathcal{A}$ delivers or removes messages in $\Pi_{\mathsf{bb-sa}}$. $\mathcal{B}$ can know this only by waiting for δ steps to see $\mathcal{A}$'s actions! Hence, $\mathcal{B}$ starts the simulation δ steps ahead of $\Pi_{\mathsf{bb-wa}}$. But, when the simulation begins, $\mathcal{B}$ doesn't yet have the real-world messages from $\Pi_{\mathsf{bb-wa}}$ that *can be copied* to $\Pi_{\mathsf{bb-sa}}$. So $\mathcal{B}$ sends non-committing TLPs to equivocate the contents of the puzzle (possible because of PROM). When $\mathcal{A}$ solves the TLP and queries the RO, actual messages from $\Pi_{\mathsf{bb-wa}}$ will be available, and $\mathcal{B}$ programs the RO to open the corresponding message from $\Pi_{\mathsf{bb-wa}}$. Since the duration between when the messages are sent and the contents learned by the honest nodes should be the same in the simulation and the real-world, we set $\Delta = 2\delta$. Thus, asymptotically, $\Pi_{\mathsf{bb-sa}}$ is round preserving (as $\Delta = 2\delta$) and communication preserving (due to compactness of our TLP).

Proof of Theorem 6.4.1. Assume that $\Pi_{\mathsf{bb-sa}}$ is *not* (δ, t)-secure protocol secure against $\mathcal{A}$, then we build adversary $\mathcal{B}$ that uses $\mathcal{A}$ to show that $\Pi_{\mathsf{bb-wa}}$ is *not* (Δ, t)-secure. At a high-level, the simulator $\mathcal{B}$:

1. Simulates honest nodes to $\mathcal{A}$ in the execution of $\Pi_{\mathsf{bb-sa}}$
2. Sends the honest messages in the simulation identical to the messages sent by the honest nodes in $\Pi_{\mathsf{bb-wa}}$.
3. Transforms every attack on $\Pi_{\mathsf{bb-sa}}$ by $\mathcal{A}$ in the simulation to an attack on $\Pi_{\mathsf{bb-wa}}$.
4. Observes the order, delay, contents of the messages sent by corrupted nodes in the simulation and replicates the same behavior in the execution of $\Pi_{\mathsf{bb-wa}}$.

$\mathcal{B}$ accomplishes this by:

- Using non-committing time-lock encryption and programming the RO to explain the contents of message while decryption;
- Using a smaller network delay parameter δ in the simulation and starting the simulation δ time-steps ahead of $\Pi_{\mathsf{bb-wa}}$.

Observe that the compiled protocol $\Pi_{\mathsf{bb-sa}}$ sends and receives same messages as $\Pi_{\mathsf{bb-wa}}$, except that the messages in $\Pi_{\mathsf{bb-sa}}$ are encrypted. Hence, it is necessary for $\mathcal{B}$ to send the honest messages in the simulation identical to the messages sent by the honest nodes in $\Pi_{\mathsf{bb-wa}}$.

Let the upper bound on the network delay in $\Pi_{\mathsf{bb-wa}}$ be $\Delta = 2\delta$ and the TLP hardness in $\Pi_{\mathsf{bb-sa}}$ be $\mathbf{T} = \delta$. Recall that $\mathcal{B}$ is as powerful as $\mathcal{A}$, *except* $\mathcal{A}$ can perform *after-the-fact* removal in the execution of $\Pi_{\mathsf{bb-sa}}$, which *cannot* be performed by $\mathcal{B}$ in the execution of $\Pi_{\mathsf{bb-wa}}$. Thus, attacks and other arbitrary behavior of $\mathcal{A}$ is translated by $\mathcal{B}$ as follows:

- **General corruptions:** Whenever, $\mathcal{A}$ corrupts a node in the simulation, $\mathcal{B}$ corrupts the corresponding node in the execution of $\Pi_{\mathsf{bb-wa}}$. Specifically, $\mathcal{B}$ hands $\mathcal{A}$: all private coins (including the secret keys) of the corrupted node. Since honest nodes in $\Pi_{\mathsf{bb-sa}}$ securely erase the internal state, $\mathcal{A}$ has to wait for T time-steps to learn the internal state of the corrupted node.
- **After-the-fact removal:** $\mathcal{B}$ handles this attack by starting the simulation δ time-steps ahead of the execution of $\Pi_{\mathsf{bb-wa}}$. Whenever $\mathcal{A}$ deletes any message in-flight sent by an honest node in $\Pi_{\mathsf{bb-sa}}$, $\mathcal{B}$ corrupts the corresponding node in $\Pi_{\mathsf{bb-wa}}$. Since the

simulation is δ time-steps ahead, the corresponding round in $\Pi_{\mathsf{bb-wa}}$ is *yet to begin.* Thus, the honest node in $\Pi_{\mathsf{bb-wa}}$ is corrupted even before it sends any messages.

Let honest denote the set of nodes currently honest in the execution of $\Pi_{\mathsf{bb-sa}}$. The simulation works as follows (Fig. 6.1 shows the simulation timeline):

For every $k \in \{0, 1, 2, \dots\}$:

- At $t = 2k\delta$:
  - If $t = 0$, $\mathcal{B}$ starts the simulation of $\Pi_{\mathsf{bb-sa}}$
  - If $t \neq 0$, every honest node in the simulation opens the $c_{v,k-1}$'s to $m_{v,k-1}$'s for all $v \in \mathsf{honest}$
  - $\mathcal{B}$ sends non-committing TLPs as:

    For every $v \in \mathsf{honest}$:

    * Pick $x_{v,k} \leftarrow \mathcal{S}$ and $c_{v,k} \leftarrow \mathcal{C}$
    * $Z_{v,k} \leftarrow \mathsf{PGen}(\mathsf{pp}, x_{v,k})$, computes $\pi_{v,k}$ as the proof of "well-formed-ness" of $Z_{v,k}$, and send $(Z_{v,k}, c_{v,k}, \pi_{v,k})$

    Since $\mathcal{A}$ is rushing, it learns all the messages instantaneously, checks $\pi_{v,k}$'s, combines all valid puzzles into a single puzzle, and starts solving the puzzle.

- At $t = (2k + 1)\delta$
  - If $t = \delta$, $\mathcal{B}$ starts the execution of $\Pi_{\mathsf{bb-wa}}$
  - As $\mathcal{B}$ is rushing, it can learn all the messages that are sent by the honest nodes in $\Pi_{\mathsf{bb-wa}}$.

- $\mathcal{A}$ delivers the messages to all honest nodes in the simulation. Also, every honest node checks $\pi_{v,k}$'s, combines all valid puzzles into a single puzzle, and starts solving the puzzle.

- Since δ time has elapsed since $\mathcal{A}$ started solving the puzzle, $\mathcal{A}$ solves the individual solutions $(s_{v,k})_{\forall v \in \mathsf{honest}}$ from the puzzle, and performs $(H(s_{v,k}))_{\forall v \in \mathsf{honest}}$.

- As $\mathcal{B}$ learnt the honest messages $(m_{v,k})_{\forall v \in \mathsf{honest}}$, it programs $H(s_{v,k})$ to return $w_{v,k}$, where $w_{v,k} = c_{v,k} \oplus m_{v,k}$.

- At every time step,
  - $\mathcal{B}$ identically maps the order, delay, contents of the messages sent by corrupted nodes in the simulation to the behavior of the corrupted nodes in execution of $\Pi_{\mathsf{bb-wa}}$.
  - If $\mathcal{A}$ corrupts an honest party or removes a message from an honest sender, $\mathcal{B}$ marks the corresponding node as corrupted in $\Pi_{\mathsf{bb-wa}}$ and transfers all private coins of the corrupted node to $\mathcal{A}$.

Using the above simulation, $\mathcal{B}$ can transform *any* attack on $\Pi_{\mathsf{bb-sa}}$ by $\mathcal{A}$ into an attack on $\Pi_{\mathsf{bb-wa}}$. From the strategy of $\mathcal{A}$, honest parties in $\Pi_{\mathsf{bb-sa}}$ determine their outputs based on the same values as what the honest parties see in the execution of $\Pi_{\mathsf{bb-wa}}$. Moreover, by construction, the honest parties infer their outputs in the execution of $\Pi_{\mathsf{bb-sa}}$ deterministically using the same output derivation g_u from $\Pi_{\mathsf{bb-wa}}$. Recall that the function g_u's are fully defined by a parties public values at the beginning of the protocol. It follows that honest parties in the execution of $\Pi_{\mathsf{bb-wa}}$ output the same as in the execution of $\Pi_{\mathsf{bb-sa}}$.

Thus, if $\Pi_{\mathsf{bb-sa}}$ is *not* (δ, t)-secure secure against $\mathcal{A}$ with probability p, then $\Pi_{\mathsf{bb-wa}}$ is also *not* (Δ, t)-secure against $\mathcal{B}$ with probability p. □

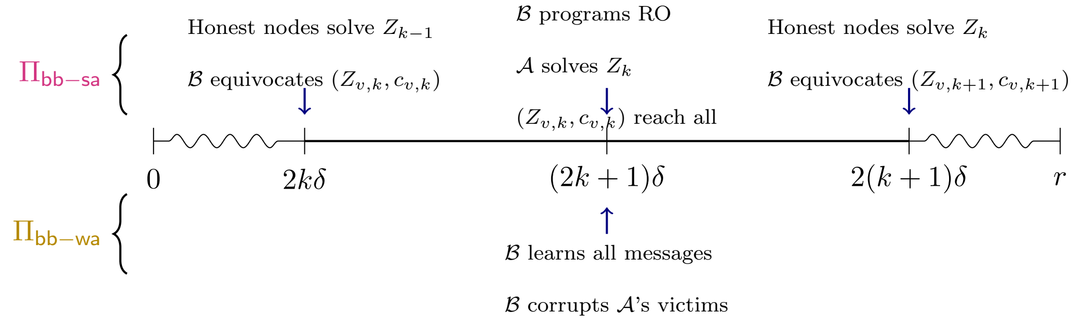


Figure 6.1: Timing of execution and messages in the simulation by $\mathcal{B}$

6.5 Related Work

Dolev and Strong showed that it is possible to have a deterministic BB protocol that terminates in $t+1$ rounds even if $t < n$ nodes are corrupt assuming PKI and a synchronous model of communication [56]. Since the seminal work by Dolev and Strong, numerous works have studied the round complexity of BB under various settings and assumptions. Notably, these approaches use randomized protocols [61] to overcome the linear round complexity barrier from [56].

Under the dishonest majority setting, prior works in BB can be categorized into: (1) weakly adaptive protocols or (2) strongly adaptive protocols. In the weakly adaptive setting, Garay et al. [68] showed the feasibility of BB in the atomic message delivery. Later, Chan et al. proposed the first expected sublinear-round complexity protocol [49]. Their protocol takes polylogarithmic (in security parameter) rounds. Subsequently, Wan et al.

proposed the state-of-the-art protocol, which takes an expected constant number of rounds to achieve BB [149]. However, the focus of our work is in the strongly adaptive setting.

In the strongly adaptive setting, Hirt and Zikas showed that it is impossible to achieve BB assuming just PKI [80]. Garay et al. [67] and Fitzi et al. proposed an expected constant-round protocol when $t = n/2 + O(1)$ [64]. Unfortunately, for any arbitrary $t = n/2 + k$, Garay et al. requires $O(k^2)$ rounds [67] and Fitzi et al. requires $O(k)$ rounds [64]. Thus, their protocols can provide expected constant-round complexity only in a narrow regime.

Subsequently, Wan et al. proposed the first expected sub-linear round protocol in the strongly adaptive setting using Public-Key Infrastructure (PKI) and TLPs [148]. Unfortunately, their protocol takes polylogarithmic number of rounds even when the $t = (1 - \varepsilon)n$, where $\varepsilon \in (0, 1)$. Subsequently, Cohen, Garay, and Zikas [53, Theorem 5] explored the feasibility of *fair* broadcast in the strongly adaptive setting for both property-based definition and simulation-based definition under different assumptions (PKI, TLP, and RO). Moreover, their work uses a *non-committing time-lock encryption* primitive to overcome the BB impossibility for a simulation based definition. However, our focus is on achieving expected constant-round BB under strongly adaptive and dishonest majority setting.

Chapter 7: Conclusion and Future Directions

7.1 Conclusion

This dissertation makes progress on two key aspects of distributed consensus: scalability and security.

First, we present authenticated data-structures to improve the scalability of distributed consensus and blockchain systems. Hyperproofs is the first vector commitment scheme that is efficiently aggregatable and maintainable. Our VC scheme is also unstealable, a novel property that incentivizes proof computation. Hyperproofs can be viewed as an algebraic variant of Merkle trees that supports efficient aggregation and unstealability. Another benefit over Merkle trees, Hyperproofs are *homomorphic*: digests (and proofs) for two vectors can be homomorphically combined into a digest (and proofs) for their sum. These properties make Hyperproofs useful in emerging applications such as stateless cryptocurrencies and beyond. For bilinear accumulators, we present a new algorithm to aggregate individual non-membership proofs into a single constant-sized proof and provide a constant-time algorithm to update individual non-membership proofs. Additionally, we evaluate new schemes to succinctly prove and verify the (non-)membership of multiple elements in a cryptographic accumulator in a privacy-preserving manner. These improvements

to bilinear accumulators are useful in emerging applications such as blockchain interoperability.

Second, we present new consensus protocols to deter network-level adversaries that can delay or delete any message in the following settings: Permissionless setting and Byzantine broadcast. Our work circumvents the prior known impossibility result in the permissionless setting using TLPs to demonstrate the feasibility of achieving secure and live consensus even after relaxing the standard synchrony assumption [114]. Furthermore, this dissertation presents an expected constant round Byzantine broadcast protocol under the strongly adaptive and corrupt majority setting. To achieve this, we develop a generic compiler to convert broadcast protocols secure against weakly adaptive adversaries into ones secure against strongly adaptive adversaries while preserving the round complexity. This compiler is of independent and practical interest.

7.2 Future directions

Vector commitment. It would be interesting to apply our aggregation and unstealability techniques to *Verkle trees* [89, 93], which are q-ary rather than binary Merkle trees. This would also help extend Hypeproofs into a key-value commitment (KVC) scheme that maps arbitrary keys to values.

Building Hyperproofs from assumptions in hidden-order groups would eliminate the large public parameters and, potentially, the trusted setup. Using more malleable inner-product arguments would allow us to update aggregated proofs too.

Last, exploring pure algebraic techniques to aggregate the individual proofs and optimizing the arguments from Figs. 3.5 and 3.8 for our Hyperproof setting could speed up aggregation and verification times as well as reduce proof size.

Accumulator. An exciting direction is to develop a technique to update the bilinear accumulator digest when an element is added to the set without using the entire set or the trapdoor. Current techniques can update the digest only if the number of elements added is the same as the number of elements deleted. Overcoming this limitation helps realize stateless validation in the UTXO setting using BP accumulators.

Permissionless consensus in the mobile sluggish model. The protocol presented in Section 5.4 did not explore parallel composition similar to prior works in the literature [12, 157]. We believe it would be interesting future work to optimize throughput and latency using such parallel composition techniques.

It is feasible to instantiate the protocol described in Section 5.4 using efficient batch TLP constructions [136, 137], albeit with less favorable asymptotic guarantees. Thus it raises the question: Is it possible to develop alternative batch TLP constructions that are both concretely efficient and better asymptotics?

Byzantine broadcast. It would be theoretically-interesting to construct an expected constant round Byzantine broadcast protocol under a strongly adaptive and corrupt majority in the *standard model*.

Bibliography

[1] Hyperledger indy website, 2015. [Online; accessed 1-May-2022].

[2] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 209–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. *Journal of Cryptology*, 29(2):363–421, Apr 2016.

[4] I. Abraham, T.-H. H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, and E. Shi. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 317–326, New York, NY, USA, 2019. Association for Computing Machinery.

[5] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 654–667. IEEE Computer Society, may 2020.

[6] T. Acar and L. Nguyen. Revocation for Delegatable Anonymous Credentials. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography – PKC 2011*, pages 423–440, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] S. Agrawal, C. Ganesh, and P. Mohassel. Non-interactive zero-knowledge proofs for composite statements. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology - CRYPTO*, volume 10993 of *Lecture Notes in Computer Science*, pages 643–673. Springer, 2018.

[8] S. Agrawal and S. Raghuraman. KVaC: Key-Value Commitments for Blockchains and Beyond. In S. Moriai and H. Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 839–869, Cham, 2020. Springer International Publishing.

[9] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2017.

[10] S. Atapoor and K. Baghery. Simulation Extractability in Groth's zk-SNARK. In C. Pérez-Solà, G. Navarro-Arribas, A. Biryukov, and J. Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 336–354, Cham, 2019. Springer International Publishing.

[11] M. H. Au, P. P. Tsang, W. Susilo, and Y. Mu. Dynamic Universal Accumulators for DDH Groups and Their Application to Attribute-Based Anonymous Credential Systems. In M. Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, pages 295–308, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 585–602, 2019.

[13] F. Baldimtsi, J. Camenisch, M. Dubovitskaya, A. Lysyanskaya, L. Reyzin, K. Samelin, and S. Yakoubov. Accumulators with Applications to Anonymity-Preserving Revocation. pages 301–315, April 2017.

[14] B. Barak, R. Canetti, Y. Lindell, R. Pass, and T. Rabin. Secure Computation Without Authentication. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 361–377, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[15] N. Barić and B. Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 480–494, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[16] P. S. L. M. Barreto and M. Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In B. Preneel and S. Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[17] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018.

[18] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014.

[19] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304, May 2015.

[20] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent Succinct Arguments for R1CS. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 103–128, Cham, 2019. Springer International Publishing.

[21] J. Benaloh and M. de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, May 1993. Springer Berlin Heidelberg.

[22] D. Benarroch, M. Campanelli, D. Fiore, K. Gurkan, and D. Kolonelos. Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular. In *Financial Cryptography and Data Security*, pages 393–414. Springer Berlin Heidelberg, 2021.

[23] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 345–356, 2016.

[24] D. Boneh and X. Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73. Springer, 2004.

[25] D. Boneh and X. Boyen. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *Journal of Cryptology*, 21(2):149–177, Apr 2008.

[26] D. Boneh, B. Bünz, and B. Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 561–586, Cham, 2019. Springer, Springer International Publishing.

[27] S. Bowe. Fast amortized Kate proofs, 2017. https://electriccoin.co/blog/new-snark-curve/.

[28] S. Bowe and J. Davies. Conclusion of the Powers of Tau Ceremony, 2018. https://www.zfnd.org/blog/conclusion-of-powers-of-tau/.

[29] S. Bowe, A. Gabizon, and M. D. Green. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In A. Zohar, I. Eyal, V. Teague, J. Clark, A. Bracciali, F. Pintore, and M. Sala, editors, *Financial Cryptography and Data Security*, pages 64–77, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

[30] S. Bowe, A. Gabizon, and I. Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model, 2017.

[31] J. Buchmann and S. Hamdy. A survey on iq cryptography. In *Public-Key Cryptography and Computational Number Theory*, pages 1–15, 2011.

[32] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the Symposium on Security and Privacy (SP), 2018*, volume 00, pages 319–338, 2018.

[33] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. Recursive Proof Composition from Accumulation Schemes. In R. Pass and K. Pietrzak, editors, *Theory of Cryptography*, pages 1–18, Cham, 2020. Springer International Publishing.

[34] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. Proofs for Inner Pairing Products and Applications. In M. Tibouchi and H. Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 65–97, Cham, 2021. Springer International Publishing.

[35] V. Buterin. The stateless client concept, Oct 2017. [Online; accessed 01-January-2022], https://ethresear.ch/t/the-stateless-client-concept/172.

[36] V. Buterin. Using polynomial commitments to replace state roots. https://ethresear.ch/t/using-polynomial-commitments-to-replace-state-roots/7095, 2020.

[37] V. Buterin. A Theory of Ethereum State Size Management, 2021. https://hackmd.io/@vbuterin/state_size_management.

[38] V. Buterin. Why Sharding is great: Demystifying the technical properties, Apr 2021. [Online; accessed 01-January-2022], https://vitalik.ca/general/2021/04/07/sharding.html.

[39] Buterin, Vitalik. The Dawn of Hybrid Layer 2 Protocols. https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html, 2019.

[40] J. Camenisch, M. Kohlweiss, and C. Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 481–500, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[41] J. Camenisch and A. Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 61–76, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[42] M. Campanelli, D. Fiore, N. Greco, D. Kolonelos, and L. Nizzardo. Vector Commitment Techniques and Applications to Verifiable Decentralized Storage. Cryptology ePrint Archive, Report 2020/149, 2020.

[43] M. Campanelli, D. Fiore, S. Han, J. Kim, D. Kolonelos, and H. Oh. Succinct zero-knowledge batch proofs for set accumulators. Cryptology ePrint Archive, Paper 2021/1672, 2021.

[44] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 2000.

[45] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable Set Operations over Outsourced Databases. In H. Krawczyk, editor, *Public-Key Cryptography – PKC 2014*, pages 113–130, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[46] D. Catalano and D. Fiore. Vector Commitments and Their Applications. In K. Kurosawa and G. Hanaoka, editors, *International Workshop on Public Key Cryptography*, pages 55–72, Berlin, Heidelberg, 2013. Springer, Springer Berlin Heidelberg.

[47] E. S. Chains. Shard chains. [Online; accessed 01-January-2022], https://ethereum.org/en/eth2/shard-chains/.

[48] B. Y. Chan and E. Shi. Streamlet: Textbook Streamlined Blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, pages 1–11, New York, NY, USA, 2020. Association for Computing Machinery.

[49] T.-H. H. Chan, R. Pass, and E. Shi. Sublinear-Round Byzantine Agreement Under Corrupt Majority. In A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, editors, *Public-Key Cryptography – PKC 2020*, pages 246–265, Cham, 2020. Springer International Publishing.

[50] A. Chepurnoy, C. Papamanthou, S. Srinivasan, and Y. Zhang. Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968, 2018. https://eprint.iacr.org/2018/968.

[51] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–768. Springer, 2020.

[52] M. Christ and J. Bonneau. Limits on revocable proof systems, with applications to stateless blockchains. In *FC '23: Proceedings of International Conference on Financial Cryptography and Data Security*, 2023.

[53] R. Cohen, J. Garay, and V. Zikas. Adaptively Secure Broadcast in Resource-Restricted Cryptography. Cryptology ePrint Archive, Report 2021/775, 2021.

[54] A. Connolly, P. Lafourcade, and O. Perez Kempner. Improved Constructions of Anonymous Credentials from Structure-Preserving Signatures on Equivalence Classes. In G. Hanaoka, J. Shikata, and Y. Watanabe, editors, *Public-Key Cryptography – PKC 2022*, pages 409–438, Cham, 2022. Springer International Publishing.

[55] I. Damgård and N. Triandopoulos. Supporting Non-membership Proofs with Bilinear-map Accumulators. Cryptology ePrint Archive, Report 2008/538, 2008.

[56] D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[57] T. Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set. Cryptology ePrint Archive, Report 2019/611, 2019.

[58] Ethereum. Shard chains, 2022. https://ethereum.org/en/upgrades/shard-chains/.

[59] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 45–59, USA, 2016. USENIX Association.

[60] D. Feist and D. Khovratovich. Fast amortized Kate proofs, 2020. https://github.com/khovratovich/Kate.

[61] P. Feldman and S. Micali. An Optimal Probabilistic Protocol for Synchronous Byzantine Agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

[62] A. Fiat and A. Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[63] Filecoin. Trusted Setup Complete!, 2020. https://filecoin.io/blog/posts/trusted-setup-complete/.

[64] M. Fitzi and J. B. Nielsen. On the Number of Synchronous Rounds Sufficient for Authenticated Byzantine Agreement. In I. Keidar, editor, *Distributed Computing*, pages 449–463, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[65] G. Fuchsbauer, E. Kiltz, and J. Loss. The Algebraic Group Model and its Applications. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 33–62, Cham, 2018. Springer International Publishing.

[66] J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer, Springer Berlin Heidelberg.

[67] J. A. Garay, J. Katz, C.-Y. Koo, and R. Ostrovsky. Round Complexity of Authenticated Broadcast with a Dishonest Majority. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, pages 658–668, Oct 2007.

[68] J. A. Garay, J. Katz, R. Kumaresan, and H.-S. Zhou. Adaptively secure broadcast, revisited. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC '11*. ACM Press, 2011.

[69] E. Ghosh, O. Ohrimenko, D. Papadopoulos, R. Tamassia, and N. Triandopoulos. Zero-Knowledge Accumulators and Set Algebra. In *Advances in Cryptology – ASIACRYPT 2016*, pages 67–100. Springer Berlin Heidelberg, 2016.

[70] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[71] go-iden3 crypto. go-iden3-crypto, 2020. https://github.com/iden3/go-iden3-crypto/.

[72] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 2007–2023, New York, NY, USA, 2020. Association for Computing Machinery.

[73] V. Goyal. Reducing Trust in the PKG in Identity Based Cryptosystems. In A. Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, pages 430–447, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[74] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[75] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 321–340. Springer, 2010.

[76] J. Groth. On the Size of Pairing-Based Non-interactive Arguments. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[77] Y. Guo, R. Pass, and E. Shi. Synchronous, with a Chance of Partition Tolerance. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 499–529, Cham, 2019. Springer International Publishing.

[78] T. Hanke, M. Movahedi, and D. Williams. DFINITY Technology Overview Series, Consensus System, 2018.

[79] L. Helminger, D. Kales, S. Ramacher, and R. Walch. Multi-party Revocation in Sovrin: Performance through Distributed Trust. In K. G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, pages 527–551, Cham, 2021. Springer International Publishing.

[80] M. Hirt and V. Zikas. Adaptively Secure Broadcast. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 466–485, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[81] Z. Hong, S. Guo, P. Li, and W. Chen. Pyramid: A Layered Sharding Blockchain System. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, May 2021.

[82] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[83] J. Katz, R. Ostrovsky, and M. O. Rabin. Identity-Based Zero-Knowledge. In C. Blundo and S. Cimato, editors, *Security in Communication Networks*, pages 180–192, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[84] D. Khovratovich and J. Law. Sovrin: digital identities in the blockchain era. *Github Commit by jasonalaw October*, 17:38–99, 2017.

[85] L. Kiffer, R. Rajaraman, and a. shelat. A Better Method to Analyze Blockchain Consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 729–744, New York, NY, USA, 2018. Association for Computing Machinery.

[86] J. Kim, V. Mehta, K. Nayak, and N. Shrestha. Making Synchronous BFT Protocols Secure in the Presence of Mobile Sluggish Faults. Cryptology ePrint Archive, Report 2021/603, 2021.

[87] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE S&P'18*, May 2018.

[88] J. Krupp, D. Schröder, M. Simkin, D. Fiore, G. Ateniese, and S. Nuernberger. Nearly Optimal Verifiable Data Streaming. In *Proceedings, Part I, of the 19th IACR International Conference on Public-Key Cryptography — PKC 2016 - Volume 9614*, pages 417–445, Berlin, Heidelberg, 2016. Springer-Verlag.

[89] J. Kuszmaul. Verkle Trees: V(ery short M)erkle Trees. In *MIT PRIMES Conference '18*, 2018.

[90] R. W. Lai and G. Malavolta. Subvector Commitments with Application to Succinct Arguments. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 530–560, Cham, 2019. Springer, Springer International Publishing.

[91] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 119–134, May 2020.

[92] J. Li, N. Li, and R. Xue. Universal Accumulators with Efficient Nonmembership Proofs. In J. Katz and M. Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[93] B. Libert and M. Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, volume 5978 of *Lecture Notes in Computer Science*, pages 499–517. Springer, 2010.

[94] H. Lipmaa. Secure Accumulators from Euclidean Rings without Trusted Setup. In F. Bao, P. Samarati, and J. Zhou, editors, *Applied Cryptography and Network Security*, pages 224–240, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[95] G. Malavolta and S. A. K. Thyagarajan. Homomorphic Time-Lock Puzzles and Applications. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 620–649, Cham, 2019. Springer International Publishing.

[96] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[97] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, May 2013.

[98] A. Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage), 2012. https://bitcointalk.org/index.php?topic=101734.msg1117428.

[99] Mitsunari Shigeo. mcl: a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl/, 2015. Accessed: 2020-10-14.

[100] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. https://bitcoin.org/bitcoin.pdf.

[101] K. Nayak, L. Ren, E. Shi, N. H. Vaidya, and Z. Xiang. Improved Extension Protocols for Byzantine Broadcast and Agreement. In H. Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[102] N. Network. How Nervos is Tackling the State Explosion Problem Facing Smart Contract Blockchains, 2021. https://medium.com/nervosnetwork/how-nervos-is-tackling-the-state-explosion-problem-facing-smart-contract-blockchains-a9acc4c5708e.

[103] L. Nguyen. Accumulators from Bilinear Pairings and Applications. In A. Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 275–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[104] A. Ozdemir. bellman-bignat, 2020. https://github.com/alex-ozdemir/bellman-bignat.

[105] A. Ozdemir, R. Wahby, B. Whitehat, and D. Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092. USENIX Association, August 2020.

[106] C. Papamanthou. *Cryptography for Efficiency: New Directions in Authenticated Data Structures*. PhD thesis, Brown University, Providence, Rhode Island, 2011. https://doi.org/10.7301/Z0Z60M99.

[107] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of Correct Computation, 2011.

[108] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of Correct Computation. In A. Sahai, editor, *Theory of Cryptography*, pages 222–242, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[109] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming Authenticated Data Structures. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 353–370, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[110] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal Verification of Operations on Dynamic Sets. In P. Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 91–110, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[111] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, May 2013.

[112] R. Pass, L. Seeman, and A. Shelat. Analysis of the Blockchain Protocol in Asynchronous Networks. In J.-S. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673, Cham, 2017. Springer, Springer International Publishing.

[113] R. Pass and E. Shi. FruitChains: A Fair Blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017.

[114] R. Pass and E. Shi. Rethinking Large-Scale Consensus. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 115–129, Aug 2017.

[115] R. Pass and E. Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 3–33, Cham, 2018. Springer International Publishing.

[116] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

[117] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. Version 0.5.9.2, https://lightning.network/lightning-network-paper.pdf.

[118] E. C. C. Prototypes and Experiments. sapling-crypto, 2021. https://github.com/zcash-hackworks/sapling-crypto.

[119] Y. Qian, Y. Zhang, X. Chen, and C. Papamanthou. Streaming Authenticated Data Structures. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security*. ACM, nov 2014.

[120] L. Ren. Analysis of Nakamoto Consensus. Cryptology ePrint Archive, Report 2019/943, 2019.

[121] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov. Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies. In A. Kiayias, editor, *Financial Cryptography and Data Security*, pages 376–392, Cham, 2017. Springer International Publishing.

[122] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.

[123] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In G. Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1989. Springer New York.

[124] J. T. Schwartz. Probabilistic algorithms for verification of polynomial identities. In *International Symposium on Symbolic and Algebraic Manipulation*, pages 200–215. Springer, 1979.

[125] S. Setty. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In D. Micciancio and T. Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 704–737, Cham, 2020. Springer International Publishing.

[126] N. Shrestha, I. Abraham, L. Ren, and K. Nayak. On the Optimality of Optimistic Responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2020.

[127] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. Cryptology ePrint Archive, Report 2021/599, 2021. https://eprint.iacr.org/2021/599.

[128] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3001–3018, Boston, MA, Aug. 2022. USENIX Association.

[129] S. Srinivasan, I. Karantaidou, F. Baldimtsi, and C. Papamanthou. Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2719–2733, New York, NY, USA, 2022. Association for Computing Machinery.

[130] S. Srinivasan, I. Karantaidou, F. Baldimtsi, and C. Papamanthou. Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. Cryptology ePrint Archive, Paper 2022/1779, 2022. https://eprint.iacr.org/2022/1779.

[131] S. Srinivasan, J. Loss, G. Malavolta, K. Nayak, C. Papamanthou, and S. A. Thyagarajan. Transparent Batchable Time-lock Puzzles and Applications to Byzantine Consensus. Cryptology ePrint Archive, Paper 2022/1421, 2022. https://eprint.iacr.org/2022/1421.

[132] S. Srinivasan, J. Loss, G. Malavolta, K. Nayak, C. Papamanthou, and S. A. Thyagarajan. Transparent batchable time-lock puzzles and applications to byzantine consensus. In *Public-Key Cryptography – PKC 2023*, pages 554–584. Springer Nature Switzerland, 2023.

[133] S.-F. Sun, M. H. Au, J. K. Liu, and T. H. Yuen. Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In *European Symposium on Research in Computer Security*, pages 456–474. Springer, 2017.

[134] S. Thakur. Batching non-membership proofs with bilinear accumulators. Cryptology ePrint Archive, Report 2019/1147, 2019. https://eprint.iacr.org/archive/2019/1147/20210929:175523.

[135] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, April 2003.

[136] S. A. K. Thyagarajan, A. Bhat, G. Malavolta, N. Döttling, A. Kate, and D. Schröder. Verifiable timed signatures made practical. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[137] S. A. K. Thyagarajan, G. Castagnos, F. Laguillaumie, and G. Malavolta. Efficient cca timed commitments in class groups. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[138] P. Todd. Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[139] A. Tomescu. *How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2020.

[140] A. Tomescu. go-mcl. https://github.com/alinush/go-mcl, 2022. [Online; accessed 1-May-2022].

[141] A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In C. Galdi and V. Kolesnikov, editors, *Security and Cryptography for Networks*, pages 45–64, Cham, 2020. Springer International Publishing.

[142] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas. Transparency Logs via Append-Only Authenticated Dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, nov 2019.

[143] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. G. Gueta, and S. Devadas. Towards Scalable Threshold Cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 877–893, May 2020.

[144] A. Tomescu, Y. Xia, and Z. Newman. Authenticated Dictionaries with Cross-Incremental Proof (Dis)aggregation. Cryptology ePrint Archive, Report 2020/1239, 2020.

[145] G. Vitto and A. Biryukov. Dynamic Universal Accumulator with Batch Update over Bilinear Groups. Cryptology ePrint Archive, Report 2020/777, 2020.

[146] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-Efficient zk-SNARKs Without Trusted Setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943, May 2018.

[147] T. Walton-Pocock. AZTEC CRS: The Biggest MPC Setup in History has Successfully Finished, 2020. https://medium.com/aztec-protocol/aztec-crs-the-biggest-mpc-setup-in-history-has-successfully-finished-74c6909cd0c4.

[148] J. Wan, H. Xiao, S. Devadas, and E. Shi. Round-Efficient Byzantine Broadcast Under Strongly Adaptive and Majority Corruptions. In R. Pass and K. Pietrzak, editors, *Theory of Cryptography*, pages 412–456, Cham, 2020. Springer International Publishing.

[149] J. Wan, H. Xiao, E. Shi, and S. Devadas. Expected Constant Round Byzantine Broadcast Under Dishonest Majority. In *Theory of Cryptography*, pages 381–411, Cham, 2020. Springer International Publishing.

[150] J. Wang and H. Wang. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association.

[151] B. Wesolowski. Efficient Verifiable Delay Functions. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 379–407, Cham, 2019. Springer International Publishing.

[152] Wikipedia contributors. The gnu multiple precision arithmetic library — Wikipedia, the free encyclopedia, 2020.

[153] Wikipedia contributors. Partial fraction decomposition — Wikipedia, the free encyclopedia, 2020. [Online; accessed 11-April-2020].

[154] Wikipedia contributors. Polynomial greatest common divisor — Wikipedia, the free encyclopedia, 2020. [Online; accessed 12-December-2020].

[155] Wikipedia contributors. Polynomial remainder theorem — Wikipedia, the free encyclopedia, 2020. [Online; accessed 03-December-2020].

[156] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf, 2014.

[157] H. Yu, I. Nikolic, R. Hou, and P. Saxena. OHIE: Blockchain Scaling Made Simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 112–127, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[158] D. H. Yum, J. W. Seo, and P. J. Lee. Generalized Combinatoric Accumulator. *IEICE - Trans. Inf. Syst.*, E91-D(5):1489–1491, May 2008.

[159] M. Zamani, M. Movahedi, and M. Raykova. RapidChain: Scaling Blockchain via Full Sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 931–948, New York, NY, USA, 2018. Association for Computing Machinery.

[160] Zcash. What is jubjub?, 2017. https://z.cash/technology/jubjub/.

[161] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE S&P 2020*, 2020.

[162] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 203–220.

[163] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880, 2017.

[164] Y. Zhang, J. Katz, and C. Papamanthou. An Expressive (Zero-Knowledge) Set Accumulator. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 158–173, April 2017.

[165] J. Zhao, J. Tang, Z. Li, H. Wang, K.-Y. Lam, and K. Xue. An Analysis of Blockchain Consistency in Asynchronous Networks: Deriving a Neat Bound. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020.

[166] R. Zippel. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Manipulation*, pages 216–226. Springer, 1979.