Plumo: An Ultralight Blockchain Client

Psi Vesely[1,2], Kobi Gurkan[2,3], Michael Straka[2], Ariel Gabizon[4], Philipp Jovanovic[2,5], Georgios Konstantopoulos[6], Asa Oines[2], Marek Olszewski[2], and Eran Tromer[2,7,8]

Abstract. Syncing the latest state of a blockchain can be a resource-intensive task, driving (especially mobile) end users towards centralized services offering instant access. To expand full decentralized access to anyone with a mobile phone, we introduce a consensus-agnostic compiler for constructing *ultralight clients*, providing secure and highly efficient blockchain syncing via a sequence of SNARK-based state transition proofs, and prove its security formally. Instantiating this, we present *Plumo*, an ultralight client for the Celo blockchain capable of syncing the latest network state summary in just a few seconds even on a low-end mobile phone. In Plumo, each transition proof covers four months of blockchain history and can be produced for just $25 USD of compute. Plumo achieves this level of efficiency thanks to two new SNARK-friendly constructions, which may also be of independent interest: a new BLS-based offline aggregate multisignature scheme in which signers do not have to know the members of their multisignature group in advance, and a new composite algebraic-symmetric cryptographic hash function.

[1]UCSD psi@ucsd.edu [2]cLabs {kobi,a,m,mstraka}@clabs.co [3]Ethereum Foundation [4]AZTEC Protocol ariel@aztecprotocol.com [5]University College London p.jovanovic@ucl.ac.uk [6]Independent Researcher me@gakonst.com [7]Columbia University [8]Tel Aviv University tromer@cs.tau.ac.il

Table of Contents

1 Introduction

Among numerous obstacles to widespread adoption of blockchain technologies, scalability has been identified as a major hurdle [Mei18]. Recent years have seen major improvements to throughput and latency via new proof-of-stake (PoS) protocols [Amo+18; Yin+19], sharding [KK+18; Al-+18], and payment channels [Mal+17; Gud+19]. This work tackles another scalability challenge: high participation costs for end users.

In order to securely interact with a blockchain without trusting a centralized party, a node must first download and verify the blockchain. The requisite data, storage, and computation resources are unavailable to many potential participants. For example, as of August 2021, the Ethereum blockchain is over 900GB (in non-archival mode). Even in light sync mode, 6.5GB of header metadata must be downloaded and verified, exceeding the bandwidth and storage available to many mobile users. Participation cost concerns for end users also apply in the context of cross-blockchain interoperability protocols, where smart contract code running on one chain (with high storage and computation costs) needs to verify the state of another chain.

High participation costs motivate the need for *ultralight* clients (UCs), which verify succinct proofs of valid blockchain data leading up to the current state. Prior attempts [Nik+17; Bon+20; Bün+20b; Che+20] have various restrictions and drawbacks, including specificity to Proof-of-Work (PoW), implementation complexity, unsuitability for smart contract blockchains, and significant blockchain performance hits outside the UC context. Some of these relative drawbacks are outlined in Table 1.

We introduce the Plumo system, an efficient UC protocol, which overcomes these drawbacks and achieves nearly-instant ultralight client synchronization. It is based on succinct transition proofs, using two new SNARK-friendly constructions.

A brief history of ultralight clients. To contextualize, we first describe previous works in more detail, and then describe how our techniques overcome prior drawbacks.

UC	proof type	consensus	SA	programmability	trusted setup	app/prover curve bits	proof sizes (days) 347	694	1,736	verifier time
Plumo	transition	BFT	✓	✓	✓	377 → 761	1.2KB	2.5KB	6.4KB	$o(n)$
Flyclient	NIPoPoW	PoW	✓	✓	χ	256	135KB	163KB	204KB	$O(\log^2 n)$
[Che+20]	transition	PoW	χ	χ	✓	753 ◡ 753	7.4KB	10KB	18KB	$o(n)$
[Che+20]	PCD	PoW	χ	χ	✓	753 ◡ 753		0.4KB		$O(1)$
Mina	PCD	Ouroboros	χ	χ	✓	753 ◡ 753		7.1KB		$O(1)$
Halo 2/Pickles	PCD	PoW/Ouroboros	χ		χ	255 ◡ 255		$O(1)$		$O(1)$

Table 1: Comparison of UCs. App curve bits denotes the size of the curve used for most network activity including making transactions; prover curve bits refers to the curve used to produce and verify UC proofs. Estimates for both [Che+20] and Flyclient proof sizes are taken from [Che+20] and are for a "barebones" (scriptless) Bitcoin. The Flyclient paper reports slightly larger proof sizes for Ethereum due to the difference in header size. Since block times for Celo are about 120× shorter than for Bitcoin, we compare UC proof sizes by time since the genesis block. Halo 2 and Pickles are both proposed network upgrades to Zcash and Mina, resp., exact proof sizes are not yet available. NIPoWPoWs are resticted to PoW networks and in particular SPV; recursive composition based PCD as used by Mina and [Che+20] requires a trusted setup; otherwise consensus, SA, programmabaility, and trusted setup should be seen as implementation choices rather than limitations of a proof type. Some proof types also impose curve requirements (see below).

Kiayias et al. introduced NIPoWPoWs in [KMZ20], a PoW-specific proof of SPV that relies on statistical properties of hashes to make probabilistic guarantees about the amount of work a chain contains. Bünz et al. extended this result in Flyclient [Bün+20b], the first

NIPoPoW-based UC, guaranteeing unconditional succinctness with $O(\log^2 n)$ sized proofs[1] and supporting variable mining difficulty. It is integrated into chains by adding Merkle Mountain Range (MMR) commitment to the transaction roots of the entire blockchain to each header. Given the latest block header containing a MMR commitment, the verifier hashes it to obtain challenge block heights pseudorandomly; they accept if also provided MMR-inclusion and subtree equality-proofs that verify with respect to those challenges and the MMR commitment.[2] Smart contracts are supported, since miners are trusted to have verified all consensus rules. However, this approach does not extend to PoS blockchains, or to full verification of a PoW blockchain, since these require checking every pertinent state transition.

Chiesa and Tromer proposed PCD, a primitive permitting distributed computations between mutually distrustful parties that run indefinitely [CT10]. Its first practical construction by Ben-Sasson et al. used recursive composition of fully succinct SNARKs over cycles of elliptic curves in [Ben+14]. Building on this PCD construction, Bonneau et al. proposed Mina (formerly known as Coda) [Bon+20], the first fully succinct (i.e., constant-sized) blockchain whose state at any time can be verified in constant time. While this results in an ideal situation for the UC verifier, these techniques impose a large performance overhead on the part of the protocol being proved (all of consensus in the case of Mina) and the heavy cryptographic machinery required imposes high development costs.

Foremost, both the UC prover and verifier, and all of the consensus verified by the UC protocol must be set over a cycle of quite inefficient pairing-friendly curves at 753 bits[3] where, e.g., it was found Groth16 verification takes roughly 15× longer than on BLS12-381 [Che+20]. Additionally, a trusted setup is required for each curve and these setups must be computed sequentially[4].

Recent developments in PCD constructions allow compatibility with transparent SNARKs and cycles of non-pairing friendly curves, which can provide 100-bits security at just 255 bits [5] . Bowe et al. introduced Halo [BGH19], later formalized as an atomic accumulation scheme by Bünz et al. in [Bün+20a]. Halo amortizes the cost of IOP [BSCS16] and AHP-based [Chi+20] proof system verification via lazy batch verification of polynomial commitment openings, recursively verifying just the comparatively cheap arithmetic checks on the evaluations. Zcash is currently working on a refinement of these techniques with "Halo 2," and Mina is introducing a "Pickles" network upgrade that will also use atomic accumulation based PCD. These advantages come at the loss of pairing-based cryptography, which powers efficiency and non-interactivity otherwise not afforded[6].

Simplifying assumptions. Using SAs provides weaker security guarantees for light clients than proving consensus in full. Adversarial control of the majority of mining power or a dishonest supermajority on a BFT committee can result in a light client being convinced of an invalid state. Under these conditions full nodes can still be convinced of an alternate

[1] The NIPowPow protocol of Kiayias et al. is forced to revert to the SPV light client protocol in the presence of bribing and selfish mining attacks.

[2] MMRs also provide an efficient mechanism to verify past transactions (see Appendix A.)

[3] MNT4-753/MNT6-753 is the most efficient known pairing-friendly cycle at 128-bits security. Evidence suggests the nonexistence of significantly better options [CCW19].

[4] Subsequent work introducing fully succinct SNARKs with universal SRSs [Mal+19] allow parallel setups, but perforance lags behind circuit-specific SNARKs [Chi+20].

[5] See, e.g., the "pasta" cycle: https://github.com/zcash/pasta.

[6] E.g., non-interactive multisignatures, used often in BFT consensus and multisignature wallets, are only possible with pairings; for consensus naive $O(n^2)$ communication can be avoided with CoSi [KK+16], but higher latency persists, and multisignature wallet spends would require participants to all be online concurrently. Pairing-based cryptography will also power Celo's forthcoming ARKE private contact discovery system (see https://celo.org/papers/future-of-digital-currencies).

history, though transactions in the malicious fork have to follow consensus rules, which can still enable a great deal of fraud and theft. The violation of such assumptions, however, would still render the blockchain insecure for full nodes, despite enabling even worse attacks for light clients. This justifies their use in practice.

Proving a light client protocol has several advantages over proving all of consensus. First, there's simply much less to prove, especially so for networks offering programmability; indeed, only Flyclient and Plumo support programmable blockchains. Even without programmability, a single prover cannot keep up with the 1tx/s Mina blockchain, and to deal with this they incentivize "SNARK workers" to compete to provide proofs for different parts of a PCD recursion tree (allowing parallelization of prover work). Second, to efficiently prove all of consensus, all of consensus must be optimized to this end. However, optimizing for SNARK arithmetization can negatively impact performance outside the context of the SNARK prover, e.g., while the BHP-BLAKE2s cryptographic hash we introduce in Section 5 is SNARK-efficient, it is much less efficient than symmetric-flavor hashes like SHA3 on conventional von Neumann computer architecture.

Transition proofs. Plumo is the first UC to use transition proofs, allowing a client hardcoded with the genesis state s_0 to sync to some later state s_n via a chain of sequential intermediate SNARKs. We believe the use of a SA is not just justified, but essential to our approach [7] ; together with heavy optimization of just the small part of consensus our light client protocol encapsulates, our SA allows each SNARK to attest to four months of blockchain history.

Our design also allows us to keep the full Celo consensus on the efficient pairing-friendly BLS12-377 curve. To get around the problem that proving signatures over the same curve they were created on is not possible without highly expensive non-native arithmetic, we borrow the approach of using a two-chain of elliptic curves introduced by Bowe et al. in Zexe [Bow+20], thus avoiding the need to run consensus over a costly pairing-friendly cycle.

Contributions. This paper presents the following contributions:

- A formal model of UCs general enough to capture all aforementioned UCs, while at the same time remaining quite simple.
- A compiler theorem capturing our simple and efficient approach to building secure UCs with transition proofs.
- BBSGLRY, a new BLS-based aggregate multisignature scheme that improves on state-of-the-art AMSP-PoP [BDN18] by removing the need to know and append the aggregate public key of one's multisignature group before signing.
- A framework for building composite algebraic-symmetric cryptographic hashes, which improve on the SNARK-efficiency of symmetric hash functions while maintaining their more well-established security guarantees, and our proposed instantiation BHP-BLAKE2s.
- A Rust implementation of Plumo showing that for $25/day USD of compute on modern cloud infrastructure an untrusted prover can provide proofs for the whole Celo network, and that a Plumo client can sync and verify a summary of the latest blockchain state in seconds even on a low-end mobile phone.

Organization. The rest of the paper is organized as follows. Section 2 gives an overview of the Plumo architecture. Section 3 describes our threat model. Section 4 presents a

[7] We believe the estimates of subsequent work [Che+20] for a transition-based UC proving full consensus of a barebones Bitcoin network to be off by an order of magnitude even assuming a circuit an order of magnitude greater than Plumo's (which required coordinating a historically large 2^{28} powers-of-τ trusted setup ceremony), and hashing with SNARK-optimized Poseidon [Gra+19]. Such circumstances would allow proofs to cover about a week, but Flyclient would offer much faster verifier time with only slightly larger proofs given the relative costs of SNARK verification and hashing.

formalization of ultralight clients, our compiler, and then Plumo as an instantiation. Section 5 presents our aggregate multisignature scheme and framework for composite algebraic-symmetric SNARK-friendly hashes, which we instantiate with Bowe-Hopwood-Pedersen and BLAKE2s. Section 6 presents benchmarks for our Plumo Rust implementation and details numerous optimizations.

We refer the reader to the appendices for additional supplemental material. Appendix A covers additional related work. Appendix B covers notation, a formal blockchain model, background on PoS and IBFT as used by Celo, and cryptographic assumptions and definitions. Appendix C describes our trusted setup ceremony and several optimizations that have enabled faster execution and verification than previous ceremonies. Appendix D includes several proofs deferred from earlier sections. Appendix E contains a discussion of Groth16 security in the presence of arbitrary oracles, including a proof that security is preserved when an oracle does not allow for breaking a paramaterized variant of discrete log. Appendix F presents a Plumo specification with details and optimizations that we omitted earlier for clarity and abstraction.

2 Overview

The Celo blockchain uses the Istanbul BFT consensus [Mon20] (see Appendix B.3). We observe that in order to verify the latest block header in BFT networks a client only needs the public keys of the current committee. As long as no committee has had a dishonest supermajority, a client who verifies a chain of committee hand-off messages certifying the PoS election results, known as *epoch messages*, does not need to check each block or even the headers of each block. Instead, to make (or verify a recent) transaction, the client simply asks for the latest (or otherwise relevant) block header, and verifies that it has been signed by a supermajority of the current committee. This constitutes the simplifying assumption (SA) and light client protocol proved by Plumo (formally, Assumption 1).

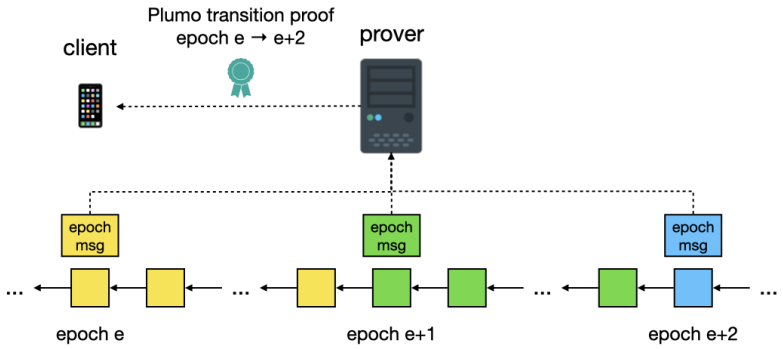

Fig. 2.1: Plumo architecture overview. In practice, our proofs cover 120 epochs.

Since Celo has 5s block times, this means transition proofs skip 17,280 blocks for every epoch message they verify. Further, it reduces the task of optimizing the transition proof SNARK circuit to just optimizing the epoch messages and their associated signatures.

In our circuit, we verify 120 sequential epoch messages, each signed by a potentially different group of roughly 67–100 validators. A multisignature is already computed over each epoch message as part of our light client protocol; compounding this efficiency, the Plumo prover aggregates these multisignatures into a single aggregate multisignature, which

costs half the constraints to verify for our BBSGLRY signature scheme. To further reduce the circuit size, instead of passing in the list of public keys that signed each epoch message, we pass in a bitmap indicating who signed, where the canonical ordering is given by the preceeding epoch message listing the committee public keys. The Hamming weight is first verified to be sufficient, and then the bitmap is used to compute the aggregate public key corresponding to each epoch message.

As cryptographic hashes that perform many bitwise operations are particularly expensive inside SNARKs, for epoch messages we instantiate BBSGLRY with a new composite cryptographic hash built from the collision-resistant Bowe-Hopwood-Pedersen hash [Hop+21] and the symmetric-flavor BLAKE2s cryptographic hash [Aum+13]. While lookup tables make it possible to at least avoid scalar multiplications, Bowe-Hopwood-Pedersen still requires many group additions, and while efficient in SNARKs is slow on conventional von Neumann computer architecture. By instantiating BBSGLRY with BLAKE2s for signing block headers, the vast majority of consensus is unaffected by this inefficiency, simultaneously ensuring ultralight clients (UCs) can efficiently verify block headers after syncing the current committee's public keys.

Aggregate multisignatures. For a longer history of BLS-based signatures, see Appendix A. The BBSGLRY aggregate multisignature scheme takes the Boneh-Lynn-Shacham (BLS) signature [BLS01] as its starting point and combines various extensions from [Bon+03; Bol03; RY07]. Its most similar to the AMSP-PoP aggregate multisignature scheme presented by Boneh et al. in [BDN18]. AMSP-PoP requires signers who create a multisignature know the group of signers in advance. In particular, signers must compute the aggregate public key apk of the signer group and then prepend it to the message before hashing and signing in the normal way: $\mathsf{Sign}(\mathsf{sk}, \mathsf{apk}, m) = \mathsf{H}_{\mathsf{s}}(\mathsf{apk} \| m)^{\mathsf{sk}}$. For one, this expands the size of our circuit by adding more data to hash. Further, this forces BFT consensus to restart if a node who participates honestly in earlier rounds goes Byzantine and fails to produce their contribution to the multisignature.

BBSGLRY overcomes these limitations as follows. We observe that in the definitions used by [BDN18] that proofs-of-possession are checked by the key aggregation algorithm KeyAgg. The adversary is permitted to output both a set of aggregate public keys and a set of pairs of public keys and PoPs. Since KeyAgg is not run on the aggregate public keys, an aggregate public key must be prepended when signing to prevent rogue key attacks. We believe their definitions do not reflect the usage of PoPs in production systems, including Celo, and have thus provided new definitions in Appendix B.5, where every public key the adversary outputs must be accompanied by a valid PoP. Working from these definitions, we are able to prove security of BBSGLRY, where signing is identical to BLS: $\mathsf{Sign}(\mathsf{sk}, m) = \mathsf{H}_{\mathsf{s}}(m)^{\mathsf{sk}}$.

SNARK-friendly hashing. When representing an arithmetic circuit in R1CS, addition gates are essentially free, while multiplication gates are not. Only recently have we seen the introduction of low-multiplication cryptographic hash functions, such as MiMC [Alb+16] and Poseidon [Gra+19]. While such hash functions are a promising development, we believe there has so far been insufficient time for cryptanalysis of these designs. As an alternative, we formalize a folklore technique of first "shrinking" a long message with an algebraic collision-resistant hash (CRH) requiring far fewer constraints per message bit, and then call the compression function of a "symmetric-flavor" cryptographic hash function on its output. Our compiler in Section 5.2 formalizes this approach and provides a security reduction appropriate for use when instantiating a random oracle (as in necessary for BBSGLRY). We instantiate our compiler with the Bowe-Hopwood-Pedersen hash and with the BLAKE2s compression function to produce the BHP-BLAKE2s cryptographic hash we use for epoch messages.

A two-chain of elliptic curves. For background on cycles and two-chains see Appendix B.4. A SNARK arithmetic circuit is defined in the scalar field $\mathbb{F}_p$ of an elliptic curve.

This presents a problem when verifying authenticated data computed over that same field, where verification (such as of BBSGLRY signatures) generally involves $\mathbb{F}_q$ operations. To avoid performing costly non-native arithmetic, which blows up circuit size, or moving to an expensive pairing-friendly cycle, we use a two-chain of elliptic curves, where the scalar field of the second curve is the same size as the base field of the first. In particular, we use the BLS12-377/BW6-761 two-chain, where the first (inner) curve is the same as in the original two-chain by Bowe et al [Bow+20], and the second (outer) was introduced by Housni and Guillevic [EHG20] as more efficient replacement for the outer curve of Bowe et al.. This allows all of consensus to be carried out over an efficient pairing-friendly curve, while only the UC prover and UC verifier when syncing use the slower second curve.

3 Threat model

In addition to a number of cryptographic hardness assumptions, Plumo makes the following security assumptions with respect to network participants:

Assumption 1. *For each epoch it holds $n > \lceil 3f/2 \rceil$, where n and f are the number of total and dishonest validators.*

Assumption 2. *There is at least a single honest participant in the multi-party computation (MPC) for the SNARK trusted setup.*

We refer the reader to Appendix B.3 for background on proof-of-stake and the Istanbul byzantine fault tolerant consensus Celo uses. There we discuss the impacts of long-range attacks and *future committee attacks*, a new related attack on PoS consensus that we identify and propose a simple defense for, on the Celo light client protocol our work builds on. For more information on the multiparty computation used for our SNARK trusted setup ceremony, including optimizations that have made it faster to carry out and verify than past public ceremonies Appendix C.

4 Ultralight clients

In Appendix B.2 we present a formal model of blockchain systems, including more explicit definitions of the building blocks which we describe below. To recap, we distinguish between full nodes, which use a state transition function S to incrementally compute the full state s corresponding to a blockchain $\boldsymbol{b} = [b_i]_{i=1}^n$ as new blocks $b_{n+1}, b_{n+2}, \ldots$ arrive, and light clients, which use the summary update function $\hat{S}$ to incrementally compute a summary $\hat{s}$ of the blockchain as they receive new trimmings $\hat{b}_{n+1}, \hat{b}_{n+2}, \ldots$. A trimming is a chunk of blockchain data (e.g., block headers for PoW blockchains or epoch messages for BFT consensus) belonging to a trimming language $\mathcal{L}_{\hat{C}}$ representing local checks such as syntax and signature verifications. A blockchain summary belongs to the summary language $\mathcal{L}_{\hat{s}}$ and is a commitment to the full state of the blockchain, enabling verification of specific transactions and full state values via succinct inclusion proofs.

Ultralight clients. Informally, we define an ultralight client (UC) to be one that receives succinct arguments of knowledge (AoKs) of trimmings. For $n \in \mathbb{Z}^+$ and $\hat{\boldsymbol{b}}$ of length n, an UC receives proofs of the *summary relation*:

$$\mathcal{R}_{\hat{s}}^{(n)} = \left\{ (\hat{s} \in \mathcal{L}_{\hat{s}};\ \hat{\boldsymbol{b}} \in \mathcal{L}_{\hat{C}}) \ :\ \hat{s} = \hat{S}(\hat{s}_g, \hat{\boldsymbol{b}}) \right\} \ .$$

An UC starts with a hardcoded genesis summary $\hat{s}_g$. It can verify $\hat{s}$ is the valid summary of the blockchain n trimmings later by verifying a succinct proof of $\mathcal{R}_{\hat{s}}^{(n)}$. The argument of

knowledge property guarantees that a valid trimmed blockchain $\hat{\boldsymbol{b}} \in \mathcal{L}_{\hat{C}}$ corresponding to $\hat{s}$ can always be extracted from the proofs a client accepts.

Incremental provers. Since prover resources are finite, for sufficiently high n it becomes impractical to prove $\mathcal{R}_{\hat{s}}^{(n)}$. An UC prover thus needs to be able to create such proofs incrementally and re-use work in some way. We model this by incrementally giving the prover one or more new trimmings each time it is invoked to create a new proof for the latest summary. The prover locally stores an auxiliary state ω to help it create the new proof. The growth of ω necessarily must be significantly sublinear in the size of the trimmed blockchain for this approach to remain concretely efficient long-term.

PCD based UCs address this by recursively verifying the previous state transition proof together with the new blocks or trimmings. Avoiding various drawbacks of this approach elaborated on in Section 1, we opt for the simpler approach of transition proofs, i.e., prove $\mathcal{R}_{\hat{s}}^{(n)}$ for any n by producing $\lceil n/m \rceil$ SNARK proofs of

$$\mathcal{R}_{\hat{s}}^{(m)} = \left\{ (\hat{s}_{i-1}, \hat{s}_i \in \mathcal{L}_{\hat{s}};\ \hat{\boldsymbol{b}} \in \mathcal{L}_{\hat{b}}^{m}) \ :\ \hat{s}_i = \hat{S}(\hat{s}_{i-1}, \hat{\boldsymbol{b}}) \right\} \ , \tag{1}$$

for $i \in \lceil n/m \rceil$. For sufficiently large n (e.g., 4 months in the case of Plumo), the concrete proof length and verification time of this sublinear approach can be on par with asymptotically better (but more complex) approaches for years out, as illustrated by our results Table 1.

Extraction in the presence of oracles. A summary relation often must some authenticated data (e.g., validator signatures). Unfortunately, standard AoK definitions fail to guarantee extraction when the adversary is granted access to additional oracles such as signature oracles. This problem has been first and foremost studied by Fiore and Nitulescu, who developed the notion of an O-SNARK and produced the first results regarding their existence [FN16]. We adapt their knowledge soundness definition to our UC interface.

4.1 Ultralight clients

An ultralight client (UC) Π_{UC} is defined by a triple of efficient non-interactive algorithms $(\mathsf{Setup}, \mathsf{ProveUpdate}, \mathsf{VerifyUpdate})$ working as follows

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$: a randomized setup algorithm run by one or more parties that, input a security parameter λ (in unary), outputs a set of public parameters pp.
- $\mathsf{ProveUpdate}(\mathsf{pp}, \hat{s}, \omega, \hat{s}', \hat{\boldsymbol{b}}) \to (\pi', \omega')$: an untrusted light client acts as the prover that, input public parameters pp, previous summary $\hat{s} \in \mathcal{L}_{\hat{s}}$ with auxiliary state ω, and current summary $\hat{s}'$ with corresponding new trimmings $\hat{\boldsymbol{b}} \in \mathcal{L}_{\hat{b}}^{n}$, outputs a new proof π and auxiliary state ω'.
- $\mathsf{VerifyUpdate}(\mathsf{pp}, \hat{s}, \pi) \to \{0, 1\}$: an UC verifier that, given a summary $\hat{s}$ and proof π, outputs 0 (reject) or 1 (accept).

and satisfying *succinctness*, *perfect completeness*, and *adaptive security*, as defined below. Assuming a strict total order $\leq$ on summaries, if presented with more than one valid $(\hat{s}, \pi)$ pair, an UC can efficiently determine and accept the greater as the current summary.

Succinctness. Let $\|\hat{\boldsymbol{b}}\|$ be the length of the description of $\hat{\boldsymbol{b}}$ (as opposed to the number of trimmings $|\hat{\boldsymbol{b}}|$). Succinctness is captured by the following set of properties:

- $|\pi|$ grows sublinearly in $||\hat{\boldsymbol{b}}||$.
- $\mathsf{VerifyUpdate}$ runs in time sublinear in $||\hat{\boldsymbol{b}}||$.
- $|\omega|$ grows sublinearly in $||\hat{\boldsymbol{b}}||$.

Completeness. An UC $\Pi_{\mathsf{UC}} = (\mathsf{Setup}, \mathsf{ProveUpdate}, \mathsf{VerifyUpdate})$ is *perfectly complete* if for every adversary $\mathcal{A}$ it holds that

$$\Pr\left[\begin{array}{c|l} \hat{\boldsymbol{b}}_1 \| \cdots \| \hat{\boldsymbol{b}}_m \in \mathcal{L}_{\hat{\mathcal{C}}} & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \wedge & [\hat{\boldsymbol{b}}_i]_{i=1}^m \leftarrow \mathcal{A}(\mathsf{pp}) \\ \exists i \in [m] : & \text{For } i \in [m] : \\ \mathsf{VerifyUpdate}(\mathsf{pp}, \hat{s}_i, \pi_i) \neq 1 & \quad \hat{s}_i \leftarrow \hat{S}(\hat{s}_{i-1}, \hat{\boldsymbol{b}}_i) \\ & \quad (\pi_i, \omega_i) \leftarrow \mathsf{ProveUpdate}(\mathsf{pp}, \hat{s}_{i-1}, \omega_{i-1}, \hat{s}_i, \hat{\boldsymbol{b}}) \end{array}\right] = 0,$$

where $\hat{s}_0 \leftarrow \hat{s}_g$, $\pi_0 \leftarrow \perp$, and $\omega_0 \leftarrow \perp$, and the probability is taken over choice of pp and any random coins used by $\mathcal{A}$. **Adaptive security.** An UC is adaptively secure if it satisfies Definition 7 for $\mathcal{R} = \mathcal{R}_{\hat{s}}^{(*)}$ and the appropriate auxiliary input generator and oracle families, and where $(\mathbb{x}, \mathbb{w}) = (\hat{s}, \hat{\boldsymbol{b}})$ and $\mathsf{Verify} = \mathsf{VerifyUpdate}$.

Flexibility of our definition. We illustrate the flexibility of our definitions by showing how they can capture PCD and NIPoWPoW based UCs as well. A trimmed blockchain can be modeled as a DAG where the current summary is the sink. Starting with the edge leaving the sole source, labeled $\hat{s}_g$, each edge $e = (\hat{s}, \hat{s}')$ is labeled with a consecutive trimming $\hat{b}$ taking the state from $\hat{s}$ to $\hat{s}' = \hat{S}(\hat{s}, \hat{b})$. Then depending on the construction of PCD used, we have $\omega = (\pi, x)$ where x is additional auxiliary information such as state tree roots and π is the proof generated by a S/NARK and/or succinct accumulator.

Next consider Flyclient [Bün+20b], where the summary is a Merkle Mountain Range commitment to the block headers, which themselves form the trimmed blockchain. Here the UC prover must store the entire trimmed blockchain on disk, but only needs to open the commitment by reading from disk block headers at a logarithmic number of heights; thus we define $|\omega|$ to be logarithmic. Here proofs, composed of leaf inclusion and subtree equality proofs, are distinct from auxiliary state, but also logarithmic in $|\hat{\boldsymbol{b}}|$.

4.2 An ultralight client compiler

We introduce a compiler that outputs a secure UC given a summary relation $\mathcal{R}_{\hat{s}}^{(m)}$ for a fixed $m \in \mathbb{Z}^+$ and O-SNARK Π_{OS} for the oracles corresponding to the authenticated data in verified $\mathcal{R}_{\hat{s}}$[8].

Construction 1. *Given a $\mathcal{Z}$-auxiliary input O-SNARK $\Pi_{\mathsf{OS}} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ for $\mathcal{R}_{\hat{s}}^{(m)}$ and for the oracle families corresponding to all data computed using a secret state verified in $\mathcal{R}_{\hat{s}}^{(m)}$, we construct an ultralight client $\Pi_{\mathsf{UC}} = (\mathsf{Setup}, \mathsf{ProveUpdate}, \mathsf{VerifyUpdate})$ as follows:*

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

$\underline{\mathsf{VerifyUpdate}(\mathsf{pp}, \hat{s}, \pi) :}$
1. Parse $([\hat{s}]_{i=1}^{k-1}, [\pi_i]_{i=1}^k) \leftarrow \pi$
2. Set $\hat{s}_0 \leftarrow \hat{s}_g$ and $\hat{s}_k \leftarrow \hat{s}$.
3. Output $b \leftarrow \wedge_{i=1}^k \mathsf{Verify}(\mathsf{crs}, \hat{s}_{i-1}, \hat{s}_i, \pi_i)$

$\underline{\mathsf{ProveUpdate}(\mathsf{pp}, \hat{s}, \omega, \hat{s}', \hat{\boldsymbol{b}})}$
1. *If $\hat{s}$ corresponds to a trimmed blockchain of n trimmings, then ω will contain $r \equiv n \bmod n$ "remainder" trimmings $\hat{\boldsymbol{b}}_r$, $k = \lceil n/m \rceil$ SNARK proofs $\boldsymbol{\pi} = [\pi]_{i=1}^k$, and $k-1$ intermediate summaries $\hat{\boldsymbol{s}} = [\hat{s}_i]_{i=1}^{k-1}$.*

[8] We note that proofs of $\mathcal{R}_{\hat{s}}^{(m')}$ for $1 \leq m' \leq m$ are called for by our construction as well. With transparent and universal setup SNARKs this can be achieved just by making m circuits, but for SNARKs with circuit-specific setups adding support for padding in $\mathcal{R}_{\hat{s}}^{(m)}$ can avoid the need for m distinct trusted setups.

2. *If* $r = 0$ *reset* $\hat{\boldsymbol{s}} \leftarrow \hat{\boldsymbol{s}} \| \hat{s}$*, else reset* $\boldsymbol{\pi} \leftarrow [\pi_i]_{i=1}^{k-1}$ *as the last proof covers only* $r < m$ *trimmings.*
3. *Set* $\hat{\boldsymbol{b}}'_1 \| \cdots \| \hat{\boldsymbol{b}}'_t \leftarrow \hat{\boldsymbol{b}}_r \| \hat{\boldsymbol{b}}$ *where partitions* $[\hat{\boldsymbol{b}}'_i]_{i=1}^{t-1}$ *each contain* m *trimmings and*
$$|\hat{\boldsymbol{b}}'_t| = r' = n + |\hat{\boldsymbol{b}}| \pmod m \quad \vee \quad m \ .$$
4. *If* $r' < m$ *then set* $\hat{\boldsymbol{b}}_{r'} \leftarrow \hat{\boldsymbol{b}}'_t$*, else set* $\hat{\boldsymbol{b}}_{r'} \leftarrow \perp$*.*
5. *Generate new intermediate states and proofs for* $i \in [t]$*:*
$$\hat{s}'_i \leftarrow \hat{S}(\hat{s}'_{i-1}, \hat{\boldsymbol{b}}'_i) \qquad \hat{\pi}_i \leftarrow \mathsf{Prove}(\mathsf{crs}, \hat{s}'_{i-1}, \hat{s}'_i;\ \hat{\boldsymbol{b}}'_i)$$
where $\hat{s}'_0$ *is the last intermediate summary in* $\hat{\boldsymbol{s}}$*.*
6. *Let* $\pi' \leftarrow \boldsymbol{\pi} \| \boldsymbol{\pi}'$*,* $\hat{\boldsymbol{s}}' \leftarrow \hat{\boldsymbol{s}} \| [\hat{s}'_i]_{i=1}^{t-1}$*, and* $\omega' \leftarrow (\hat{\boldsymbol{b}}_{r'}, \pi', \hat{\boldsymbol{s}}')$*. Output* (π', ω')*.*

In Appendix D we prove the following adaptive security theorem.

Theorem 1. *If* $\Pi_{\mathsf{OS}} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *is an adaptively secure SNARK for relation* $\mathcal{R}_{\hat{s}}$*, auxiliary input generator* $\mathcal{Z}$*, and oracle family* $\mathbb{O}$*, then the UC* Π_{UC} *output by Construction 1 is adaptively secure (Section 4.1) for* $\mathcal{R}_{\hat{s}}$*,* $\mathcal{Z}$*, and* $\mathbb{O}$*.*

4.3 The Plumo ultralight client

We make a few simplifications for clarity of exposition in this section; a full specification of our circuit is present in Appendix F. Celo uses the Istanbul BFT consensus algorithm [Mon20]. We observe that by taking Assumption 1 as our simplifying assumption (SA), a light client only needs verify a valid chain of epoch messages delegating authority from committee to the next in order to learn the current committee public key set. From there, they can download the most recent block header, verify its multisignature, and learn the latest state roots (and also easily check their balance, make a transaction, etc.).

The most recent Celo epoch message is the current summary. In addition to the current committee public key set, the summary contains the epoch index, the *current and parent entropy* (see future committee attacks Appendix B.3), and the signer threshold[9]. The standard operator $\leq$ over the epoch index of each summary defines the required total order $\leq$ over summaries (a strict total order under our simplifying assumption).

The summary update relation checks there exists a sequence of epoch messages where each successive message (1) is signed by at least the signer threshold number of validators, (2) increases the epoch index by 1, and (3) has parent entropy matching the previous current entropy. Then it verifies an aggregate multisignature over the result.

Plumo instantiates the compiler from the previous section using the Groth16 proof system, which was proven to be knowledge sound in the AGM under the q-DLOG assumption in [FKL18]. For Plumo, we must additionally require Groth16 is an O-SNARK with respect to BBSGLRY signing oracles . We also assume that the auxiliary input our adversary receives is "benign"[10]. We note here that there have been few prior results on extraction in the presence of auxiliary inputs and/or oracles [Bit+16; FN16], none of which apply to our construction[11].

Theorem 2. *Let* $\mathcal{H} : \{0,1\}^* \to \mathbb{G}_1$ *be a hash family modeled as a random oracle and let* $\mathsf{BBSGLRY}_{\mathcal{H}}$ *be the* BBSGLRY *signature scheme (Section 5.1) instantiated with* $\mathcal{H}$*,* [12] *and*

[9] Our PoS election occasionally elects n<100 committee members. Rather than compute $\lceil 2n/3 \rceil + 1$ in the circuit, we piggyback on our SA, including it in the epoch message.

[10] A benign distribution supplies negligible advantage to any adversary against any construction (e.g., the uniform distribution is conjectured benign [Bit+13]).

[11] Results for hash-then-sign signatures in [FN16] require modifying the signer to sample and prepend a random nonce to each message they sign—currently no UCs which prove verification of signatures are doing this.

[12] A single hash family $\mathcal{H} : \{0,1\}^* \to \mathbb{G}_1$ can instantiate both $\mathcal{H}_{\mathsf{s}} : \{0,1\}^* \to \mathbb{G}_1$ and $\mathcal{H}_{\mathsf{p}} : \mathbb{G}_2 \to \mathbb{G}_1$ required by BBSGLRY given an injective coding $\mathsf{Encode} : \mathbb{G}_2 \to \{0,1\}^*$.

let $\mathcal{Z}$ be a benign auxiliary input generator. Assume the Groth16 *SNARK is an adaptive argument of knowledge (Definition 7) for $(\mathcal{O}_{\mathcal{H}}, \mathcal{O}_{\mathsf{BBSGLRY}_{\mathcal{H}}})$ and $\mathcal{Z}$. Then* Plumo *is an adaptively secure UC for $\mathcal{R}_{\hat{s}}$, $\mathcal{Z}$, $\mathcal{O}_{\mathcal{H}}$, and $\mathcal{O}_{\mathsf{BBSGLRY}_{\mathcal{H}}}$.*

Proof. This follows directly from the compiler Theorem 1. □

5 SNARK-friendly signatures and hashing

5.1 BBSGLRY: non-interactive aggregate multisignatures

BBSGLRY [13] is an offline aggregate multisignature scheme providing non-interactive key and signature aggregation, and not requiring signers know the multisignature group in advance.

Construction 2 (BBSGLRY aggregate multisignature scheme)**.** *Given a type 3 bilinear group sampler* $\mathsf{SampleGrp}_3$ *and two hash families $\mathcal{H}_{\mathsf{s}} : \{0,1\}^* \to \mathbb{G}_1$ and $\mathcal{H}_{\mathsf{p}} : \mathbb{G}_2 \to \mathbb{G}_1$, our aggregate multisignature scheme* BBSGLRY *is defined by an 8-tuple of efficient algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{VPoP}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{AggSign}, \mathsf{Verify})$*, working as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$*: sample a type 3 bilinear group* $\langle \mathsf{group} \rangle \leftarrow \mathsf{SampleGrp}_3(1^\lambda)$ *and two hash functions* $(\mathsf{H}_{\mathsf{p}}, \mathsf{H}_{\mathsf{s}}) \xleftarrow{\$} \mathsf{H}_\lambda$*. Return* $\mathsf{pp} \leftarrow (\langle \mathsf{group} \rangle, \mathsf{H}_{\mathsf{p}}, \mathsf{H}_{\mathsf{s}})$.
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{pk}, \mathsf{sk}, \pi)$*: choose a secret key* $\mathsf{sk} \xleftarrow{\$} \mathbb{F}$ *and set the public key* $\mathsf{pk} \leftarrow G_2^{\mathsf{sk}} \in \mathbb{G}_2$*. Create the PoP* $\pi \leftarrow \mathsf{H}_{\mathsf{p}}(\mathsf{pk})^{\mathsf{sk}} \in \mathbb{G}_1$*. Return* $(\mathsf{pk}, \mathsf{sk}, \pi)$.
- $\mathsf{VPoP}(\mathsf{pp}, \mathsf{pk}, \pi)$*: given public key* $\mathsf{pk} \in \mathbb{G}_2$ *and PoP* $\pi \in \mathbb{G}_1$*, return 1 if* $e(\pi,\ G_2) = e(\mathsf{H}_{\mathsf{p}}(\mathsf{pk}),\ \mathsf{pk})$*, else 0.*
- $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m) \to \sigma$*: given a secret key* $\mathsf{sk} \in \mathbb{F}$ *and message* $m \in \{0,1\}^*$*, return a signature* $\sigma \leftarrow \mathsf{H}_{\mathsf{s}}(m)^{\mathsf{sk}} \in \mathbb{G}_1$.
- $\mathsf{KeyAgg}(\mathsf{pp}, \{\mathsf{pk}_i\}_{i=1}^n) \to \mathsf{apk}$*: given n distinct public keys* $\{\mathsf{pk}_i\}_{i=1}^n \in \mathbb{G}_2^n$*, return aggregrate public key* $\mathsf{apk} \leftarrow \prod_{i=1}^n \mathsf{pk}_i \in \mathbb{G}_2$.
- $\mathsf{MultiSign}(\mathsf{pp}, \{\sigma_i\}_{i=1}^n) \to \sigma$*: given n signatures* $\{\sigma_i\}_{i=1}^n \in \mathbb{G}_1^n$ *under distinct public keys for the same message, return multisignature* $\sigma \leftarrow \prod_{i=1}^n \sigma_i \in \mathbb{G}_1$.
- $\mathsf{AggSign}(\mathsf{pp}, [\sigma_i]_{i=1}^n) \to \Sigma$ *: given a list of n multisignatures* $[\sigma_i]_{i=1}^n \in \mathbb{G}_1^n$*, return aggregate multisignature* $\Sigma \leftarrow \prod_{i \in [n]} \sigma_i \in \mathbb{G}_1$.
- $\mathsf{Verify}(\mathsf{pp}, [(\mathsf{apk}_i, m_i)]_{i=1}^n, \Sigma) \to \{0,1\}$ *: given a list of n aggregate public key and message pairs* $[(\mathsf{apk}_i, m_i)]_{i=1}^n$ *and an aggregate multisignature* Σ*, return 1 if* $e(\Sigma,\ G_2) = \prod_{i=1}^n e(\mathsf{H}_{\mathsf{s}}(m_i),\ \mathsf{apk}_i)$*; else return 0.*

In Appendix D we prove the following unforgeability theorem.

Theorem 3. BBSGLRY *is a computationally unforgeable aggregate multisignature (Definition 6) under ψ-co-CDH (Definition 3) when instantiated with random oracles $\mathsf{H}_{\mathsf{s}}, \mathsf{H}_{\mathsf{p}}$.*

5.2 Composite algebraic-symmetric hash functions

BHP-BLAKE2s is a cryptographic hash function that first "shrinks" its input using the SNARK-optimized Bowe-Hopwood-Pedersen (BHP) collision-resistant hash [Hop+21], then runs the BLAKE2s compression function [Aum+13] on the result. We prove security via instantiating the following construction.

[13] Pronounced "BBS glory" and named after the authors whose work it incorporates and extends. See "A history of BBSGLRY" in Appendix A for more details.

Construction 3. *Given collision-resistant hash* $\mathsf{CRH} : \{0,1\}^* \to \mathcal{B}$ [14] *, injective encoding* $\mathsf{Encode} : \mathcal{B} \to \{0,1\}^{b-t}$*, and random oracle* $\mathcal{O} : \{0,1\}^b \to \{0,1\}^c$ *for positive integers* ℓ *and* $t \geq \lceil \log_2(\lceil \ell/c \rceil + 1) \rceil$*, we construct a composite hash function* $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ *as follows. Let* $k \leftarrow \lceil \ell/c \rceil$*, and for integers* $0 \leq x \leq 2^t - 1$ *denote by* xut *the* t*-bit unsigned binary representation of* x*. On input* $m \in \mathcal{M}$*:*

1. *Shrink the message to obtain the intermediate hash* $h' \leftarrow \mathsf{CRH}(m)$*.*
2. *Compute the binary encoding of the intermediate hash* $h'_{\mathsf{enc}} \leftarrow \mathsf{Encode}(h')$*.*
3. *Output the first* ℓ *bits of* $\mathcal{O}(\mathsf{0ut} \| h'_{\mathsf{enc}}) \| \mathcal{O}(\mathsf{1ut} \| h'_{\mathsf{enc}}) \| \ldots \| \mathcal{O}(\mathsf{kut} \| h'_{\mathsf{enc}})$*.*

In Appendix D we prove the following indistinguishability theorem.

Theorem 4. *If* CRH *is computationally collision-resistant (Definition 8),* Encode *is injective, and* $\mathcal{O}$ *is a random oracle, then the hash function* H *is computationally indistinguishable from a random oracle.*

In BHP, presented below, input messages are split into segments m_i, then further divided into 3-bit chunks $m_{i,j}$. The maximum number of chunks in a segment, denoted C_{max}, depends on the curve. A formula to derive it is given in [Hop+21].

$\underline{\mathsf{BHP.Setup}(1^\lambda, s) \to \mathsf{pp}}$	$\underline{\mathsf{BHP.Eval}(\mathsf{pp}, m \in \{0,1\}^n) \to h}$
$(\mathbb{G}, q) \leftarrow \mathsf{SampleGroup}(1^\lambda)$	Divide m into segments m_i of size C_{max}
$[g_i]_{i=1}^s \leftarrow \mathbb{G}^s$	Divide each m_i into 3-bit chunks $m_{i,j}$
$\mathsf{pp} \leftarrow (\mathbb{G}, q, [g_i]_{i=1}^s)$	$h \leftarrow \sum_{i,j} g_i^{2^{4i}(1+m_{i,j}[0]+2 \cdot m_{i,j}[1])(1-2 \cdot m_{i,j}[2])}$

We refer the reader to [Aum+13] for a description of the BLAKE2s.

6 Implementation

Plumo was implemented in Rust[15] using the arkworks [16] libraries. In Section 6.1 we discuss additional optimizations we implemented, and in Section 6.2 we present some benchmarks illustrating its concrete efficiency.

6.1 Optimizations

Try-and-increment hashing. Since constant-time hashing is not important to the security of Plumo, we opt for a more efficient hash-to-group by using a variant of "try-and-increment" [BLS01]. For a Weierstrass form curve, let q be the order of the base field and $\ell = \lceil \log_2(q) \rceil$. Given a hash function $\mathsf{H} : \{0,1\} \to \{0,1\}^{\ell+1}$ and input m, we can hash to $\mathbb{G}_1$ using rejection sampling as follows. Try each sequential nonce η in $0, \ldots, 2^c - 1$ encoded as c-bit string (for some completeness parameter c) until the first ℓ bits of $h \leftarrow \mathsf{H}(\eta \| m)$ is less than q. To obtain a prime-order group point from h, clear the cofactors from the first ℓ bits of h to obtain an x-coordinate. If the last bit of h is 0 (1) choose the smaller (larger) corresponding y-coordinate.

We crucially observe that it is not necessary to increment inside the SNARK, and that the nonce can be included as a private input. Indeed, if we write the message of any signature scheme as $\mathcal{M} = \{0,1\}^c \times \mathcal{M}'$, where $\mathcal{M}'$ is considered the meaningful part, then the

[14] The codomain $\mathcal{B}$ may be, e.g., a group $\mathbb{G}$.

[15] See https://github.com/celo-org/celo-bls-snark-rs and https://github.com/celo-org/snark-setup.

[16] https://github.com/arkworks-rs

unforgeability of a signature on any message in $\mathcal{M}$ implies the unforgeability of a signature on any message in $\mathcal{M}'$.

In the ROM, the probability of succeeding on each try is $q/2^\ell$, and thus an expected $2^\ell/q$ tries will be required to hash each message. The chance a given message cannot be hashed is given by $(1 - q/2^\ell)^c$. For our concrete parameters, BLS12-377 and $c = 8$, this gives an exceedingly small 2^{-677} probability a message cannot be hashed.

Computing BHP over a birationally equivalent curve. Following [Hop+21], we compute the Bowe-Hopwood-Pedersen hash over the birationally equivalent Montgomery form of the twisted Edwards curve $E_{\mathsf{Ed/BW6}}$ curve (of equal order to BW6-761) in a way that guarantees the incomplete addition formulas (which cost 3 constraints instead of 6) are sufficient.

Batched Miller loops. Verifying a BBSGLRY aggregate multisignature over m messages requires computing $m+1$ pairings. A pairing consists of computing a Miller loop ML followed by a final exponentiation FE. We use the well-known optimization of computing the Miller loops in parallel, taking the product of the Miller loops, and finally computing a single final exponentiation on the product, checking the equivalent verification equation:

$$\mathsf{FE}(\mathsf{ML}(\Sigma, G_2^{-1}) \cdot \mathsf{ML}(\mathsf{H_s}(m_1), \mathsf{apk}_1) \cdots \mathsf{ML}(\mathsf{H_s}(m_m), \mathsf{apk}_m)) \stackrel{?}{=} 1_{\mathbb{G}_T} \quad . \tag{2}$$

Reducing verifier time and proof sizes. Verification of Groth16 requires computing a $\mathbb{G}_1$ multi-exponentiation of size $\ell = |\mathbb{x}|$. If the initial and m-epochs-later epoch messages were directly encoded as the instance, ℓ would be approaching 1,000. Instead, the verifier hashes the input and output epoch messages using a hash-to-field built with BLAKE2s, producing an input and output hash, which is the instance of size $\ell = 2$ for the Groth16 verification circuit. The circuit has to be modified to prove knowledge of openings of these two hashes, and then the usual checks are made on these openings. This unfortunately increases the size of the circuit, but at least this cost is constant in the number of epochs being proved.

This optimization gives us another for free. The ultralight client (UC) only needs to learn the most recent epoch message. When verifying multiple SNARK proofs the UC can simply download the intermediate summaries as hashes, thereby significantly reducing proof sizes.

Finally, the UC uses batch verification when verifying multiple Groth16 proofs. Let n be the number of proofs being verified. We use a variant of the small exponent test [BGR98; CL06] to reduce a naive $3n$ pairings to $n + 2$ and then compute only a single final exponentiation as in Eq. (2).

6.2 Evaluation

We benchmarked our prover on a Google Cloud machine with 4 Intel Xeon E7-8880 v4 processors and 3,844GB of DDR4 RAM, which rents for $25/h USD. Fig. 6.1 shows the time and space efficiency of our prover, and Table 3 gives our circuit size as a function of the committee size and number of epochs spanned. Since proofs for 120 epochs are computable in less than an hour and epochs are approximately one day, maintaining up-to-date UC proofs for Plumo is possible for $25 worth of compute a day.

In contrast to our powerful prover, we evaluated the performance of our verifier on a Motorola Moto X (2nd Gen), a 2014 mobile phone with 1GB RAM and a 32-bit Quad-core 2.45 GHz Krait 400 processor. We used a directly cross-compiled, unoptimized implementation. The results show it is possible to verify such a proof in about 0.5 seconds.

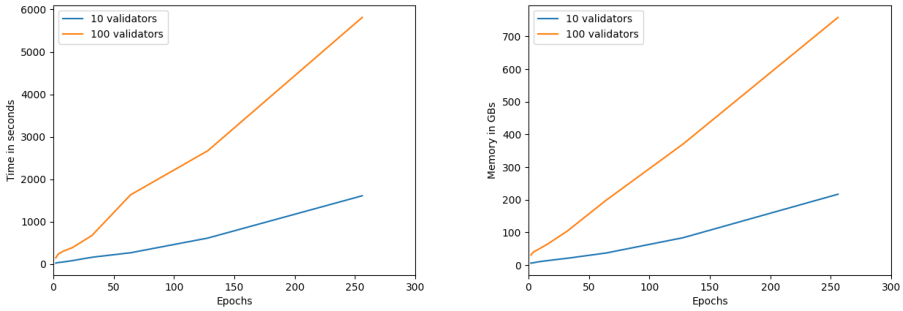

Fig. 6.1: Proving time and peak memory consumption over BW6-761.

Epochs	10 validators	100 validators
32	2,787,485	20,465,083
64	4,753,568	34,097,470
128	8,685,734	61,362,244
256	16,550,063	115,891,789
512	32,278,721	224,950,879
1024	63,736,037	443,069,059

Table 3: Constraints for our summary update transition proof circuit.

A Additional related work

Transaction inclusion. Flyclient [Bün+18] introduces a new mechanism for efficient proofs of transaction inclusion. A new datastructure called a Merkle Mountain Range (MMR) is added to the block header, whose leaves are sequentially updated with the transaction root of each new block. Then with just the latest block header and a Merkle inclusion proof, an ultralight client can efficiently confirm any transaction.

Recall that finality is not immediate on PoW blockchains, and it takes approximately an hour to get a new transaction mined with an adequate number of child blocks (confirmations) to be trustworthy. In the interim, a Flyclient client would not have a trustworthy inclusion proofs, and would thus have to download and verify every transaction starting from the oldest input to the present, which is impractical. TICK [Zha+20] solves this problem by additionally adding a UTXO tree to the block header as well. They observe using an AVL hash tree [AVL62] the commitment can be efficiently updated in $O(M \cdot \log(N_U))$ time, where M is the total number of inputs and outputs in a block and N_U is the total number of UTXOs.

TxChain [Zam+20] introduces a protocol enabling greater efficiency for verifying large numbers of transactions by introducing contingent transactions. Given a list of transactions $[t_i]_{i=1}^n$, a prover makes a new transaction t_a that references $[t_i]_{i=1}^n$. By verifying t_a an ultralight client is convinced of the validity of $[t_i]_{i=1}^n$. This approach may be particularly useful for cross-chain interoperability, as verifying transactions in a smart contract can be especially costly.

While TICK is generally not applicable to PoS blockchains, which offer immediate finality, all these techniques potentially offer complementary functionality to an ultralight client built with our consensus-agnostic compiler.

Layer Two Scaling. Layer 1 (L1) solutions are baked into the blockchain's consensus rules, while layer 2 (L2) are built on top of the underlying protocol (e.g., using its scripting/programmatic features) and so are easier to iterate on. Several L2 solutions have been proposed which work across both PoW and PoS protocols. Plasma and TrueBit both rely on fraud proofs, by which actors are normally assumed to be honest but with a mechanism to affect their punishment through economic disincentives should they show malicious behavior [PB17]. The approach most similar to ours is ZK Rollup, where every change to the state is accompanied by a SNARK proof attesting to its validity, created by the block proposer. Optimistic rollups[17] are similar, but each new state is initially assumed to be true with the caveat that fraud proofs can be submitted to slash the node submitting the new state if it is false.

Other Light Client Approaches. Other approaches for blockchain scalability have been proposed which do not fit into the above categories. The Tendermint Light Client [Bra+20], built on Tendermint Core [Amo+18], is a BFT consensus algorithm where at least 1/3rd of validators are assumed to be correct in "trusting periods"—a limited time window after they sign. A light client observing that a guaranteed correct validator by this assumption in a sufficiently recent block $n - m$ where $m < n$ signed block n containing a commit for block $n - 1$, may then assume block $n - 1$ is correct and skip downloading it when verifying the chain. This approach, however, is not necessarily compatible with verifying multiple blocks in a single cryptographic proof as we describe in this work.

CoVeR [CKK20] describes a different protocol which enables light clients to collaboratively validate blocks without assuming those blocks are validated by full nodes. When a block is broadcast to the network, light clients query for random portions of the block. Honest light clients then produce fraud proofs for invalid block portions. A light client determines

[17] https://medium.com/matter-labs/optimistic-vs-zk-rollup-deep-dive-ea141e71e075

the validity of a block by the presence or absence of such fraud proofs. Assuming a small minority of honest light clients, this guarantees no required trust assumptions with respect to full nodes, at the cost of increased light client computation and communication costs, in addition to larger block sizes. By comparison, PLUMO requires assuming the existence of an honest minority of full nodes, but works to minimize light client computation costs.

A history of BBSGLRY. The development of BBSGLRY starts with Boneh-Lynn-Shacham (BLS) signatures [BLS01]. BLS was extended to support signature aggregation by Boneh et al. in [Bon+03]. Their scheme only supports aggregate signatures on distinct messages because otherwise a rogue key attack is possible. The authors note that by prepending their own public key to each message signers can ensure their messages are distinct, but extending this technique to aggregate multisignatures (e.g., AMSP [BDN18]) requires signers know the multisignature group in advance. This increases consensus complexity and requires hashing all public key in the multisignature group, making computation inside a SNARK impractical for large committees.

Multisignature support was added to BLS by Boldyreva in [Bol03]. Her scheme was set in the knowledge-of-secret-key (KOSK) model, where the adversary must output a corresponding secret key for each public key. This precludes rogue-key attacks, allowing aggregate public keys and multisignatures to be computed as simple products, and signers to sign simply the message alone without prepending the multisignature group key set.

The KOSK abstracts the PoP as something proved sound independently, but as shown by Ristenpart and Yilek it is necessary to prove joint security [RY07]. Our scheme incorporates the B-PoP protocol from [RY07].

Finally, the aggregate multisignature AMSP-PoP, introduced in [BDN18], combines the above-mentioned works as well. Signatures for their scheme require prepending the aggregate public key to the message. We show that this restriction is unnecessary and signers can sign as in BLS. This is accomplished by changes to the interface and definitions we believe better reflect real-world use, most pertinently that the adversary in our definition must output a valid PoP for every public key. This in particular prevents rogue-key attacks (see Appendix B.5 for more details).

B Preliminaries

B.1 Notation

We denote by $[n]$ the set $\{1, \ldots, n\} \subseteq \mathbb{N}$. When x and y are two vectors, we denote the concatenation of $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_m)$ by $x \,|\, y = (x_1, ..., x_n, y_1, ..., y_m)$. We use $\boldsymbol{a} = [a_i]_{i=1}^n$ as a short-hand for the vector $(a_1, \ldots, a_n)$, and $[\boldsymbol{a}_i]_{i=1}^n = [[a_{i,j}]_{j=1}^m]_{i=1}^n$ as a short-hand for the vector $(a_{1,1}, \ldots, a_{1,m}, \ldots, a_{n,1}, \ldots, a_{n,m})$; $|\boldsymbol{a}|$ denotes the number of entries in $\boldsymbol{a}$. We analogously define $\{a_i\}_{i=1}^n$ with respect to sets instead of vectors. If x is a binary string then $|x|$ denotes its bit length. For a finite set S, let $x \xleftarrow{\$} S$ denote that x is an element sampled uniformly at random from S. We sometimes use Python-like slicing where $\boldsymbol{a}[i : j]$ is the subvector containing $(i + 1)$-th through j-th entries of $\boldsymbol{a}$, and $\boldsymbol{a}[i]$ denotes the $(i + 1)$-th entry of $\boldsymbol{a}$.

NP Relations. We write $\{(\mathbb{x}; \mathbb{w}) : p(\mathbb{x}, \mathbb{w})\}$ to describe a NP relation $\mathcal{R} \subseteq \{0,1\}^* \times \{0,1\}^*$ between instances $\mathbb{x}$ and witnesses $\mathbb{w}$ decided by the polynomial-time predicate $p(\cdot, \cdot)$.

Security notions. We denote by $\lambda \in \mathbb{N}$ a security parameter. When we state that $n \in \mathbb{N}$ for some variable n, we implicitly assume that $n = \text{poly}(\lambda)$. We denote by $\text{negl}(\lambda)$ an unspecified function that is *negligible* in λ (namely, a function that vanishes faster than the inverse of any polynomial in λ). When a function can be expressed in the form $1 - \text{negl}(\lambda)$, we say that it is *overwhelming* in λ. When we say that algorithm $\mathcal{A}$ is an *efficient* we mean that

$\mathcal{A}$ is a family $\{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ of non-uniform polynomial-size circuits. If the algorithm consists of multiple circuit families $\mathcal{A}_1, \ldots, \mathcal{A}_n$, then we write $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$.

B.2 Blockchain model

We present a formal model of blockchain systems, with a focus on the aspects necessary to define ultralight clients in Section 4.1.

Consensus language. The consensus algorithms of a blockchain system define a polynomial-time *consensus language* $\mathcal{L}_C$. If $\boldsymbol{b} \in \mathcal{L}_C$, then we say $\boldsymbol{b}$ is a *valid blockchain*, where a blockchain is a finite vector of one or more blocks $\boldsymbol{b} = (b_1, b_2, \ldots)$. Consensus algorithms also imply a *block language* $\mathcal{L}_b$, representing local checks such as syntax and signature verifications. A block b is a *valid block* if $b \in \mathcal{L}_b$. A valid blockchain must contain only valid blocks, but the converse may not hold, i.e., blocks in a blockchain may be individually valid but mutually inconsistent.

Consensus algorithms also define an efficiently computable binary relation $\leq$ that is a strict total order be defined on the set $\mathcal{L}_C$, i.e., for every $\boldsymbol{b}, \boldsymbol{b}' \in \mathcal{L}_C$ either $\boldsymbol{b} < \boldsymbol{b}'$, $\boldsymbol{b} > \boldsymbol{b}'$, or $\boldsymbol{b} = \boldsymbol{b}$. For example, for Bitcoin the chain with more work is greater and for Celo the chain that is longer.

State transition function. As each new block that extends a blockchain is incrementally proposed, it would be impractical to have to run a $\mathcal{L}_C$ membership predicate on the full blockchain. All practical blockchain systems thus implicitly define a notion of a chain *state* s, that is sublinear in the length of the chain, but contains all the necessary information to efficiently decide if each new block is valid and then produce the next state of the chain.

We thus define the consensus language of a blockchain in terms of a *state transition function* $S : \mathcal{L}_{\hat{s}} \times \mathcal{L}_b \to \mathcal{L}_{\hat{s}} \cup \{\bot\}$ that, given a state corresponding to a blockchain it has already verified and a new block, outputs an updated state if the new block is a valid extension to the blockchain, or $\bot$ otherwise. Denote the set of valid blockchains by $\mathcal{L}_C = \{\boldsymbol{b} \in \mathcal{L}_b \mid S(s_g, \boldsymbol{b}) \neq \bot\}$, where s_g is the genesis state and we use the syntactic sugar $S(s, \boldsymbol{b}) = S(\cdots S(S(s, b_1), b_2) \cdots, b_n)$. The *state language* $\mathcal{L}_{\hat{s}}$ is simply defined as all states reachable by valid blockchains.

Simplifying assumptions and summaries. Often by making certain reasonable assumptions it is possible to compute the state of a blockchain (or just a commitment to it) more efficiently. For example, the simplified payment verification (SPV) assumption used by many PoW blockchain light clients assumes "the chain with the most PoW solutions follows the rules of the network and will eventually be accepted by the majority of miners" [Bün+20b].

We refer to the information a light client learns as a *summary* $\hat{s}$ of the state s. A summary may not always be enough to fully verify or interact with the blockchain in every way, but it should be enough to facilitate access to most functionality not immediately available through efficient interactions with helper full nodes that may, e.g., provide succinct transaction inclusion proofs or act as a server for PIR.

We formalize simplifying assumptions and summaries analogously to consensus and state. We begin by introducing a *trim* function T that maps a blockchain $\boldsymbol{b}$ in the consensus language to its *trimmed blockchain* counterpart $\hat{\boldsymbol{b}}$ in the *simplified consensus language* $\mathcal{L}_{\hat{C}}$. We first define a *trim* function $T : \mathcal{L}_b \to \mathcal{L}_{\hat{b}}$ that takes as input a valid block and outputs a smaller *trimming* $\hat{b}$ in the *trimming language* $\mathcal{L}_{\hat{b}}$. The trimming language, like the block language represents local checks such as syntax and signature verifications. Under a reasonable simplifying assumption, a light client that verifies $\hat{\boldsymbol{b}} \in \mathcal{L}_{\hat{C}}$ can have confidence there exists a $\boldsymbol{b} \in \mathcal{L}_C$ such that $\hat{\boldsymbol{b}} = T(\boldsymbol{b})$. We write $T(\boldsymbol{b}) = (T(b_1), T(b_2), \ldots)$. We require that when T is applied to any valid blockchain, the resulting trimmed blockchain is accepted by $\hat{S}$, described below.

Analogous to the state transition function, the *summary update function* $\hat{S} : \mathcal{L}_{\hat{s}} \times \mathcal{L}_{\hat{b}} \to \mathcal{L}_{\hat{s}} \cup \{\perp\}$ takes a summary $\hat{s}$ corresponding to a trimmed blockchain it has already verified and a new trimming $\hat{b}$, and outputs an updated summary $\hat{s}'$ if (under the simplifying assumption) $\hat{b}$ corresponds to a block that presents a valid extension to a blockchain that produced summary $\hat{s}$, or else outputs $\perp$. Hence, $\mathcal{L}_{\hat{C}} = \left\{ \hat{\boldsymbol{b}} \in \mathcal{L}_{\hat{b}} \mid \hat{S}(\hat{s}_g, \hat{\boldsymbol{b}}) \neq \perp \right\}$. The *state language* $\mathcal{L}_{\hat{s}}$ is simply defined as all states reachable by valid blockchains.

Lastly, we require a strict total order $\leq$ on summaries such that $\hat{s} \leq \hat{s}'$ implies the existence of corresponding blockchains $\boldsymbol{b} \leq \boldsymbol{b}'$.

B.3 Proof-of-stake consensus

Celo validators are elected by a proof-of-stake voting mechanism. Once elected, they serve on a committee for one epoch, which is currently set to 24 hours worth of blocks. Celo validators trade off proposing and confirming blocks using the Istanbul Byzantine Fault Tolerant (IBFT) consensus algorithm [Mon20]. IBFT is deterministic, assumes a partially synchronous communication model, and guarantees safety independent of timing assumptions when $n > \lceil 3f/2 \rceil$, where n and f are the number of total and byzantine nodes. BFT consensus protocols provide the following guarantees [CGR11]:

- Termination: Every correct replica eventually decides some value v.
- Validity: If all replicas propose the same value v, then no replica decides a value different from v; a correct replica may only decide a value that was proposed by some correct replica or the special value $\perp$ indicating that no valid decision was found.
- Integrity: No correct replica decides twice.
- Agreement: No two correct replicas decide differently.

Compared to Nakamoto consensus, which is based on proof-of-work and the heaviest-chain rule, proof-of-stake and BFT-based consensus is fork-free given $n > \lceil 3f/2 \rceil$. In Nakamoto consensus, even with a honest majority, forks may occur regularly as miners find blocks at the same time, or because of attacks like selfish mining. With Bitcoin, this means it is common practice to require 6 confirmation blocks, waiting roughly an hour, to ensure the finality—that a transaction has really once and for all been included in the blockchain.

Long range attacks. BFT blockchains must incentivize honesty and prevent double-signing, the signing of two different blocks at the same height, in order to have a single chain. Positive behavior is financially rewarded, while double signing is punished with slashing, where a validator's stake is taken. Since stake is not locked forever, past validators who don't have locked stake anymore, can choose to collude and extend a chain built on top of an older block without fear of losing their stake. We refer to such forks as long-range attacks, which can cause light clients who only verify part of a blockchain to accept a fork that does not follow consensus rules (whereas full nodes may accept a fork, but only one that followed consensus rules). Celo employs a combination of incentives (e.g., maintaining a reputation as an honest validator and continuing to earn block rewards) and security protocols (e.g., key rotation, checkpointing) as a defense against such attacks.

Future committee attacks. Consider a scenario where at some point an adversary $\mathcal{A}$ obtains signing oracle access to a number of validators (during possibly disjoint periods), who later form the supermajority of some committee. When blocks are predictable, the adversary can use each validator key it gains access to sign blocks for an adversarial public key set and for every index out to decades in the future. So even if the adversary no longer has oracle access to any of the validators for that epoch, they will be able to create a fork. We call this a *future committee attack* (although it applies to non-commitee based PoS networks as well), which to the best of our knowledge has not been described in the prior literature.

Two possible defenses against such an attack include a high water mark for block number implemented in trusted hardware (which could alert the validator to their compromise) and

regularly enforced key rotations. PLUMO has opted for another solution that doesn't rely on trusted hardware or place additional burden on validators: we have included the verifiable randomness value from the last block of the current ("current entropy") and previous epoch ("parent entropy") in each epoch message. This makes the epoch messages unpredictable and thus not signable in advance, as an adversary would have to guess the randomness from the epoch before the epoch they would otherwise be able to fork from.

B.4 Cryptographic assumptions

Bilinear groups The cryptographic primitives that we construct in this paper rely on cryptographic assumptions about bilinear groups. We formalize these via a *bilinear group sampler*, which is an efficient algorithm $\mathsf{SampleGrp}$ that given a security parameter λ (represented in unary), outputs a tuple $\langle \mathsf{group} \rangle = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, G_1, G_2, e)$ where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are groups with order divisible by the prime $q \in \mathbb{N}$, G_1 generates $\mathbb{G}_1$, G_2 generates $\mathbb{G}_2$, and $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a (non-degenerate) bilinear map.

Following [GPS08], we distinguish between three types of bilinear group samplers. Type I groups have $\mathbb{G}_1 = \mathbb{G}_2$ and are known as *symmetric* bilinear groups. Types II and III are *asymmetric* bilinear groups, where $\mathbb{G}_1 \neq \mathbb{G}_2$. Type II groups have an efficiently computable homomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$, while Type III groups do not have an efficiently computable homomorphism in either direction. Certain assumptions are provably false w.r.t. certain group types (e.g., ψ-co-CDH only holds for Type III groups), and in general in this work we assume we are working with Type III groups.

Chains of elliptic curves Let E be an elliptic curve over a finite field $\mathbb{F}_q$, where q is a prime. We denote this by $E/\mathbb{F}_q$, and we denote by $E(\mathbb{F}_q)$ the group of points of E over $\mathbb{F}_q$, with order $n = \#E(\mathbb{F}_q)$. We say that an elliptic curve $E/\mathbb{F}_q$ is *pairing-friendly* if $E(\mathbb{F}_q)$ has a large prime-order subgroup, and if the embedding degree (i.e., the smallest integer k such that n divides q^{k-1}) is small.

Definition 1 (Two-chain of elliptic curves). *A two-chain of elliptic curves is a pair of distinct elliptic curves $E_1/\mathbb{F}_{q_1}, E_2/\mathbb{F}_{q_2}$, where q_1, q_2 are prime, such that $\#E_1(\mathbb{F}_{q_1}) = q_2$.*

We say an elliptic curve is *ordinary* if $E[q] \equiv \mathbb{Z}/q\mathbb{Z}$, where $[q]$ is the multiplication-by-p map.

Definition 2 (Pairing-friendly two-chain). *A (k_1, k_2)-chain is a two-chain of distinct ordinary elliptic curves $E_1/\mathbb{F}_{q_1}, E_2/\mathbb{F}_{q_2}$ with respective embedding degrees k_1, k_2. A (k_1, k_2)-chain is pairing-friendly if k_1 and k_2 are small.*

A 2-chain of elliptic curves $E_1/\mathbb{F}_{q_1}, E_2/\mathbb{F}_{q_2}$ where $\#E_1(\mathbb{F}_{q_1}) = q_2$ is useful as it allows for the computation of elliptic curve operations and pairings for E_2 inside of an arithmetic circuit defined over the scalar field $\mathbb{F}_{q_2}$ of E_1.

Cryptographic assumptions The computational ψ-co-Diffie-Hellman assumption was introduced in [BDN18]. For type 1 and 2 pairings it is equivalent to co-CDH, and it is assumed to hold whenever co-CDH does.

Definition 3 (Computational ψ-co-Diffie-Hellman (ψ-co-CDH) [BDN18]). *For a bilinear group sampler $\mathsf{SampleGrp}$, let $\psi(\cdot)$ be an oracle that on input $G_2^\gamma \in \mathbb{G}_2$ outputs $G_1^\gamma \in \mathbb{G}_1$. We say ψ-co-CDH is hard with respect to $\mathsf{SampleGrp}$ if for all efficient adversaries $\mathcal{A}$ it holds that*

$$\Pr\left[y = G_1^{\alpha\beta} \;\middle|\; \langle \mathsf{group} \rangle \leftarrow \mathsf{SampleGrp}(1^\lambda);\;\; \alpha, \beta \xleftarrow{\$} \mathbb{F};\;\; y \leftarrow \mathcal{A}^{\psi}(\langle \mathsf{group} \rangle, G_1^\alpha, G_1^\beta, G_2^\beta)\right] \leq \mathrm{negl}(\lambda) \;\; .$$

Definition 4 (q-DLog). *For a bilinear group sampler* SampleGrp*, we say q-DLog is hard with respect to* SampleGrp *if for all efficient adversaries* $\mathcal{A}$ *it holds that*

$$\Pr\left[y = x \;\middle|\; \langle \mathsf{group}\rangle \leftarrow \mathsf{SampleGrp}(1^\lambda);\;\; x \xleftarrow{\$} \mathbb{F};\;\; y \leftarrow \mathcal{A}(\langle \mathsf{group}\rangle, G_1^x, G_1^{x^2}, ..., G_1^{x^q}, G_2^x, G_2^{x^2}, ..., G_2^{x^q})\right] \leq \mathrm{negl}(\lambda)\ .$$

For any cryptographic hardness assumption involving only one adversary $\mathcal{A}$, we also introduce the concept of $\mathbb{O}$-*hardness* for arbitrary oracle families $\mathbb{O}$. For such a cryptographic assumption, we say that this assumption is $\mathbb{O}$-hard if it holds when $\mathcal{A}$ is replaced with $\mathcal{A}^{\mathcal{O}}$ where $\mathcal{O} \xleftarrow{\$} \mathbb{O}$, with the randomness in the assumption's probability statement being taken also over the choice of $\mathcal{O}$.

B.5 The algebraic group and random oracle models

We follow [Chi+20] in our definition of the algebraic group model (AGM), first introduced in [FKL18]. In the AGM, all algorithms are modeled as *algebraic*, which means that whenever an algorithm outputs a group element G_1, the algorithm must also output an "explanation" of G_1 in terms of the group elements that it has seen.

Definition 5 (algebraic algorithm). *Let* $\mathbb{G}$ *be a cyclic group of prime order* q *and* $\mathcal{A}_{\mathsf{alg}}$ *a probabilistic algorithm run on initial inputs including description* $\langle \mathsf{group}\rangle$ *of* $\mathbb{G}$*. During its execution* $\mathcal{A}_{\mathsf{alg}}$ *may interact with oracles or other parties and receive further inputs including obliviously sampled group elements (which it cannot sample directly*[18]*). Let* $\boldsymbol{L} \in \mathbb{G}^n$ *be the list of all group elements* $\mathcal{A}_{\mathsf{alg}}$ *has been given so far such that all other inputs it has received do not depend in any way on group elements*[19]*. We call* $\mathcal{A}_{\mathsf{alg}}$ *algebraic if whenever it outputs a group element* $G \in \mathbb{G}$ *it also outputs a vector* $\boldsymbol{a} = [a_i]_{i=1}^n \in \mathbb{F}_q^n$ *such that* $G = \sum_{i=1}^n a_i L_i$*. The coefficients* $\boldsymbol{a}$ *are called the "representation" of* G *with respect to* $\boldsymbol{L}$*, denoted* $G := \langle \boldsymbol{a}, \boldsymbol{L}\rangle$*.*

Aggregate multisignatures Our definition of an aggregate multisignature scheme is based on [BDN18], but we make several changes. First, since the BLS-based scheme we use in Plumo has a non-interactive signing process, we have simplified the interface of the Sign algorithm accordingly. Further, we define only a single verification algorithm, noting that (multi)signatures are just aggregate (multi)signatures with a single message, and that signatures are just multisignatures with signer group size one. Lastly, since we want to prove joint security in the plain public key model, we include a proof-of-possession (PoP) scheme as part of the interface, where PoP generation is folded into KeyGen and PoP verification is handled by a new algorithm VPoP. An aggregate multisignature scheme then consists of a 8-tuple of efficient algorithms (Setup, KeyGen, VPoP, Sign, KeyAgg, MultiSign, AggSign, Verify) that behave as follows:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$: a setup algorithm that, given a security parameter λ (represented in unary), outputs a set of public parameters pp.
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{pk}, \mathsf{sk}, \pi)$: a key generation algorithm that outputs a public-secret key pair $(\mathsf{pk}, \mathsf{sk})$ and a PoP π.

[18] Outputting obliviously sampled group elements (with unknown representation) is forbidden in the AGM. Instead, $\mathcal{A}_{\mathsf{alg}}$ must obliviously sample elements through an additional oracle $\mathcal{O}$ such that they are by definition added to the list $\boldsymbol{L}$. Simulating $\mathcal{O}$ to an algebraic algorithm during a reduction is straightforward and always possible. Integrating the ROM and AGM indeed works for this reason that any outputs from random oracles are added to the list $\boldsymbol{L}$.

[19] The restriction that all inputs to algebraic algorithms that are *not* group elements must not depend on group elements helps to avoid pathological cases. For example, the algorithm that on input "$G\|0$" (which is not a group element), outputs group element G cannot explain G in terms of previously seen group elements.

- $\mathsf{VPoP}(\mathsf{pp}, \mathsf{pk}, \pi) \to \{0,1\}$: a PoP verification algorithm that, given a public key pk and a corresponding PoP π, returns 1 or 0 to accept or reject the proof, respectively.
- $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m) \to \sigma$: a signing algorithm that, given a secret key sk and message $m \in \{0,1\}^*$, returns a signature σ.
- $\mathsf{KeyAgg}(\mathsf{pp}, \{\mathsf{pk}_i\}_{i=1}^n) \to \mathsf{apk}$: a key aggregation algorithm that, given a set of n public keys $\{\mathsf{pk}_i\}_{i=1}^n$, returns an aggregate public key apk.
- $\mathsf{MultiSign}(\mathsf{pp}, \{\sigma_i\}_{i=1}^n) \to \sigma$: a non-interactive multisignature algorithm that, given n signatures $\{\sigma_i\}_{i=1}^n$) (on the same message under distinct keys), returns a multisignature σ.
- $\mathsf{AggSign}(\mathsf{pp}, [\sigma_i]_{i=1}^n) \to \Sigma$: a non-interactive aggregate multisignature algorithm that, given a list of n (multi)signatures, outputs an aggregate signature Σ.
- $\mathsf{Verify}(\mathsf{pp}, [(\mathsf{apk}_i, m_i)]_{i=1}^n, \Sigma) \to \{0,1\}$: an aggregate multisignature verification algorithm that, given a list of public key and message pairs $[(\mathsf{pk}_i, m_i)]_{i=1}^n$ and an aggregate multisignature Σ, returns 1 or 0 to accept or reject the signature, respectively.

We require that an aggregate multisignature scheme satisfies unforgeability. Our unforgeability definition is based on [BDN18], but deviates in an important way: namely, for every public key the adversary outputs, they must also output a corresponding valid PoP. We believe this to be a practical and widely standard-in-practice assumption for a system using PoPs. Further, it allows us to prove unforgeability of our aggregate multisignature scheme BBSGLRY in Section 5.1 under the same assumptions as AMSP-PoP from [BDN18].

BBSGLRY is nearly identical to AMSP-PoP (including in their mutual use of the PoP B-PoP from [RY07]), but unlike AMSP-PoP does not require signers prepend the aggregate public key of thes multisignature to their messages and thus know the multisignature group before signing. Appending the aggregate public key is unnecessary in practice for their scheme, but forced by their definitions and interface. PoPs are not checked by the unforgeability challenger, but instead by the KeyAgg algorithm of AMSP-PoP. Their adversary also outputs the aggregate public keys which do not need to contain the challenge public key directly, instead of outputting them as public key sets as in our definition. This means KeyAgg is never run on them and hence if the aggregate public key was not prepended to the message when signing rogue-key attacks would be possible.

Changing their definition to just check PoPs on the aggregate public keys output by the adversary would not capture the same guarantee, since this would preclude the possibility of an adversary who can only produce a forgery that is checked against one or more aggregate public keys (in addition to the one that must contain the challenge public key) that they cannot produce PoPs for directly, but for which they can produce PoPs for corresponding sets of public keys which when passed to KeyAgg result in those aggregate public keys.

We can see their unforgeability definition as a subcase of our own, where the adversary outputs public key sets $[\mathcal{PK}_i]_{i=1}^n$ of size one, and where VPoP is set to the constant 1 function. Further, since signatures and multisignatures can be seen as subcases of aggregate multisignatures as noted above, our unforgeability definition covers all three.

Definition 6 (Unforgeable aggregate multisignature). *For an aggregate multisignature scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{VPoP}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{AggSign}, \mathsf{Verify})$ *we define the advantage of an adversary against unforgeability to be defined by* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{forge}}(1^\lambda) = \Pr\left[\mathsf{Game}_{\mathcal{A}}^{\mathsf{forge}}(1^\lambda) = 1\right]$ *where the game* $\mathsf{Game}_{\mathcal{A}}^{\mathsf{forge}}$ *is defined as follows for* $n \in \mathbb{N}$.

$\underline{\mathsf{Game}^{\mathsf{forge}}_{\mathcal{A}}(1^\lambda)}$
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\mathsf{pk}^*, \mathsf{sk}^*, \pi^*) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
$Q \leftarrow \emptyset$
$(\{(\mathcal{PK}_i, m_i)\}_{i=1}^n,\ \{\Pi_i\}_{i=1}^n,\ \mathcal{PK},\ \Pi^*,\ m^*,\ \Sigma) \leftarrow \mathcal{A}^{\mathsf{Sign}}(\mathsf{pp}, \mathsf{pk}^*, \pi^*)$
If $\mathsf{pk}^* \notin \mathcal{PK}^* \ \vee\ m^* \in Q$ *then return* 0
For $i \in [n]$ *:*
  For $(\mathsf{pk}, \pi) \in \mathcal{PK}_i \times \Pi_i$ *:*
    If $\mathsf{VPoP}(\mathsf{pp}, \mathsf{pk}, \pi) = 0$ *then return* 0
  $\mathsf{apk}_i \leftarrow \mathsf{KeyAgg}(\mathsf{pp}, \mathcal{PK}_i)$
For $(\mathsf{pk}, \pi) \in \mathcal{PK}^* \times \Pi^*$ *:*
  If $\mathsf{VPoP}(\mathsf{pp}, \mathsf{pk}, \pi) = 0$ *then return* 0
$\mathsf{apk}^* \leftarrow \mathsf{KeyAgg}(\mathsf{pp}, \mathcal{PK}^*)$
Return $\mathsf{Verify}(\mathsf{pp}, [(\mathsf{apk}_i, m_i)]_{i=1}^n \| (\mathsf{apk}^*, m^*)], \Sigma)$

$\underline{\mathsf{Sign}(m)}$
$\sigma \leftarrow \mathsf{Sign}(\mathsf{pp}, \mathsf{sk}^*, m)$
$Q \leftarrow Q \cup \{m\}$
Return σ

We say an aggregate multisignature scheme is unforgeable if for all efficient adversaries $\mathcal{A}$ it holds that $\mathsf{Adv}^{\mathsf{forge}}_{\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda)$.

O-SNARKs: SNARKs in the presence of oracles In this section we introduce the notion of an O-SNARK [FN16], which is a SNARK that allows for knowledge extraction in the presence of oracles.

Definition 7 ($\mathcal{Z}$-auxiliary input O-SNARK for $\mathbb{O}$). *A $\mathcal{Z}$-auxiliary input succinct non-interactive argument of knowledge for the oracle family $\mathbb{O}$ and the relation $\mathcal{R}$ is a triple of efficient algorithms $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ working as follows*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{crs}$*: on input of a security parameter λ (expressed in unary), outputs a common reference string* crs.
- $\mathsf{Prove}(\mathsf{crs}, \mathbb{x}, \mathbb{w}) \to \pi$*: given a common reference string* crs*, an instance $\mathbb{x}$, and a witness $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, this algorithm produces a proof π.*
- $\mathsf{Verify}(\mathsf{crs}, \mathbb{x}, \pi) \to \{0, 1\}$*: on input of a common reference string* crs*, an instance $\mathbb{x}$, and a proof π, the verifier algorithm outputs 0 (reject) or 1 (accept).*

and satisfying perfect completeness, succinctness, and adaptive argument of knowledge specified as follows:

- *Perfect Completeness: For every $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ it holds that:*

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

- *Succinctness: For every $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$ and $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ it holds that:*
  - *$|\pi| = \mathrm{poly}(\lambda)$, where $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, \mathbb{x}, \mathbb{w})$ (i.e., proof size is defined by a universal polynomial in the security parameter λ), and*
  - Verify *runs in time* $\mathrm{poly}(\lambda + |\mathbb{x}|)$.
- *Adaptive argument of knowledge: Π satisfies adaptive argument of knowledge for $\mathbb{O}$ and $\mathcal{Z}$ if for every efficient oracle prover $\mathcal{A}^{\mathcal{O}}$ who makes at most $Q(\lambda) = \mathrm{poly}(\lambda)$ queries there exists an efficient extractor $\mathcal{E}_{\mathcal{A}}$ with black box access to $\mathcal{A}$ including any random coins such that:*

$$\Pr\left[\begin{array}{c} \mathsf{Verify}(\mathsf{crs}, \mathbb{x}, \pi) = 1 \\ \wedge \\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{r} \mathsf{aux} \leftarrow \mathcal{Z};\ \ \mathcal{O} \leftarrow \mathbb{O};\ \ \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathbb{x}, \pi) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{crs}, \mathsf{aux}) \\ \mathbb{w} \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{crs}, \mathsf{aux}, \mathsf{qt}) \end{array}\right] \leq \mathrm{negl}(\lambda) \ ,$$

  where $\mathsf{qt} = \{(q_i, \mathcal{O}(q_i))\}$ is the query transcript of all queries and answers made and received by $\mathcal{A}^{\mathcal{O}}$.

Hash functions Below we give a definition of a collision-resistant hash family with a key space $\mathcal{K}$, message space $\mathcal{M}$, and codomain $\mathcal{Y}$. We note that elsewhere in this work we often omit discussion of key sampling for simplicity, and since for functions like BLAKE2s that has already been done and fixed in advanced.

Definition 8 (Collision-resistance). *Let $\mathcal{H}_\lambda : \mathcal{K}_\lambda \times \mathcal{M} \to \mathcal{Y}_\lambda$ be a hash function family. We say $\mathcal{H}$ is computationally collision-resistant if for all efficient adversaries $\mathcal{A}$*

$$\Pr\left[k \leftarrow \mathcal{K}_\lambda;\ (m_0, m_1) \leftarrow \mathcal{A}(k) \mid m_0 \neq m_1\ \wedge\ \mathsf{H}_k(m_0) = \mathsf{H}_k(m_1)\right] \leq \mathrm{negl}(\lambda)\ .$$

C Trusted setup

Fully succinct SNARKs, including the Groth16 SNARK used by Plumo require a trusted party to compute a structured reference string (SRS) used for both proof generation and verification. To avoid centralizing trust, we use secure multi-party computation (MPC) to perform a distributed online trusted setup where participants from around the world are encouraged to contribute. During such a ceremony multiple participants individually generate pieces of randomness—sometimes called *toxic waste*—which they use to perform their part of the MPC and then delete afterwards. This process has the strong security guarantee that only one honest participant needs to delete their toxic waste after finishing their contribution [20] . Each participant generates proofs to show they performed their part of the MPC correctly, which can also be used to verify their contribution is part of the final SRS. Our trusted setup ceremony builds on the "MMORPG" protocol introduced by Bowe et al. and used by Zcash [BGM17] and "Snarky Ceremonies" protocol by Kohlweiss et al. [Koh+21]. We augment these ceremonies with an "optimistic setup," allowing more efficient contribution by a set of participants who can be added on a rolling basis to an ongoing ceremony, and a combination of batch verification techniques reducing MPC verification time to a small fraction of the time it takes naively.

Optimistic setup. Consider generating two pieces of randomness $\alpha = \alpha_0 \cdot \alpha_1 \cdot \alpha_2 \cdot ... \cdot \alpha_n$ and $\beta = \beta_0 \cdot ... \cdot \beta_n$, where each α_i and β_i is generated by some participant i in a trusted setup ceremony. The most direct way to achieve this is to have participant $i-1$ pass its resulting contributions $\alpha'_{i-1} = \alpha_0 \cdot ... \alpha_{i-1}$ and β'_{i-1} computed analogously on to participant i, who will then compute $\alpha'_i = \alpha'_{i-1} \cdot \alpha_i$ and similarly β'_i, before then giving both to participant $i+1$ who will then iteratively repeat the same process.

This approach, while valid, carries with it two distinct problems. First, it requires for maximum efficiency that participant i is available immediately after participant $i-1$. Secondly, it is inefficient, even when run with a minimum of downtime.

A solution to both of these problems, which we have implemented [21], is known as *optimistic out-of-order execution*[22]. Consider the above example where each α and β represent vectors of some large size n of random elements, so that computing one vector takes several hours. Observe that when participant i is computing vector α_i, no progress is being made on the β vector. We could instead have two different participants work on each vector simultaneously. In fact, if we have n participants, we could split each vector into $m \geq n$ *chunks*, so that each participant can each work on a chunk simultaneously, before giving it back to some untrusted server which will then give it to another participant after it has finished contributing to its chunk. In addition to the gains from parallelism, this has the benefit that the setup can be split into *rounds* in which a subset of participants only need to be online together for a

[20] Join over 100 participants so far in our ongoing ceremony at https://celo.org/plumo .
[21] See our open-source implementation: https://github.com/celo-org/snark-setup-operator.
[22] https://ethresear.ch/t/accelerating-powers-of-tau-ceremonies-with-optimistic-pipelining/6870

relatively short period of time, after which a new round of a distinct subset of participants can contribute to the previous round's output pending an arbitrary duration between them. This has the added benefit of making it easy to add new participants to the setup in a rolling basis.

To show security of such a scheme over the base MMORPG scheme, it suffices to show that the proof of knowledge of exponent used in the protocol is secure in the face of certain auxiliary inputs, in particular the CRS elements computed by and received from other participants. This is in fact shown by [Koh+21], in which they prove security of the proof of exponent protocol against an adversary able to make oracle queries to obtain random evaluations on arbitrary Laurent polynomials, effectively simulating being able to see partially computed elements of the CRS.

Batch verification. We use a combination of the bucket and small exponent test as described by Bellare et al. in [BGR98] to significantly reduce the number of pairings needed to verify our trusted setup ceremony. Benchmarks confirm these techniques provide almost a 50× speedup in verification over a naive approach.

Security. Cheon showed that when given G_1, G_1^{α}, $G_1^{\alpha^i}$ for any power $i \mid q-1$, where q is the prime order of $\mathbb{G}_1$, it is possible to find the DL α in time $O(\sqrt{q/i} + \sqrt{i})$ [Che10]. Using the Pollard-Rho variant[23] of Cheon's attack (which is even faster than the original baby-step giant-step based variant) we can lower bound the number of $\mathbb{G}_1$ exponentiations an adversary would have to perform to $1.25(\sqrt{q/i} - \sqrt{i})$. Therefore, despite the very large 2^{28} size of our trusted setup, we estimate the security over BW6-761 to be at least 175 bits.

D Deferred proofs

Proof of Theorem 1.

Proof. For every efficient oracle adversary $\mathcal{A}^{\mathcal{O}}$ that on input $(\mathsf{pp}, \mathsf{aux})$, with non-negligible probability outputs $(\hat{s}, \pi = ([\hat{s}]_{i=1}^{c-1}, [\pi_i]_{i=1}^{c}))$ such that the VerifyUpdate algorithm of Π_{UC} accepts. We define an efficient extractor $\mathcal{E}_{\mathcal{A}}$ with negligible knowledge soundness error $\kappa(\lambda)$ that, on input $(\mathsf{pp}, \mathsf{aux}, \mathsf{qt})$ and the random coins of $\mathcal{A}$, outputs $\boldsymbol{b}$ such that $\hat{S}(\hat{s}_g, \boldsymbol{b}) = \hat{s}$.

Assume $\mathcal{A}$ produces an accepting summary-proof pair. Then

$$\wedge_{i=1}^{c} \mathsf{Verify}(\mathsf{crs}, \hat{s}_{i-1}, \hat{s}_i, \pi_i) \ ,$$

where $\hat{s}_0 \leftarrow \hat{s}_g$ and $\hat{s}_c \leftarrow \hat{s}$. Let $\mathcal{B}_i^{\mathcal{O}}$ be the O-SNARK oracle adversary that on input $(\mathsf{crs}, \mathsf{aux})$ runs $\mathcal{A}$ on $(\mathsf{crs}, \mathsf{aux})$ and its own local random tape, relaying the oracle queries of $\mathcal{A}$ to its own oracle(s), and outputs $((\hat{s}_{i-1}, \hat{s}_i), \pi_i)$ taken from the output of $\mathcal{A}$. By assumption of the adaptive security of Π_{OS}, there exists an efficient extractor $\mathcal{E}_{\mathcal{B}_i}$ with negligible knowledge error $\kappa'(\lambda)$ that, on input $(\mathsf{crs}, \mathsf{aux}, \mathsf{qt})$ and the random coins of $\mathcal{B}_i$, with non-neglible probability outputs $\hat{\boldsymbol{b}}_i$ such that $\hat{S}(\hat{s}_{i-1}, \hat{\boldsymbol{b}}_i) = \hat{s}_i$.

By running extractors $\mathcal{E}_{\mathcal{B}_1}, \ldots, \mathcal{E}_{\mathcal{B}_c}$ on its own inputs $(\mathsf{pp}, \mathsf{aux}, \mathsf{qt})$, the extractor $\mathcal{E}_{\mathcal{A}}$ obtains $\hat{\boldsymbol{b}} \leftarrow \hat{\boldsymbol{b}}_1 \| \cdots \| \hat{\boldsymbol{b}}_c$. It follows from the union bound that the knowledge soundness of the ultralight client is $\kappa(\lambda) = c \cdot \kappa'(\lambda)$, where the negligible function $\kappa'(\lambda)$ gives the knowledge soundness of Π_{OS}. Since $\mathcal{A}$ is efficient, $c = \mathrm{poly}(\lambda)$, implying both that $\kappa(\lambda)$ is negligible and that $\mathcal{E}_{\mathcal{A}}$, which runs in c calls to $\mathcal{E}_{\mathcal{B}_i}$, is efficient, as required. □

Proof of Theorem 3.

[23] https://ethresear.ch/t/cheons-attack-and-its-effect-on-the-security-of-big-trusted-setups/6692

Proof. Given a $(\tau, q_{\mathsf{S}}, q_{\mathsf{H_s}}, \epsilon)$ forger $\mathcal{F}$ against $\mathsf{BBSGLRY}$, we build a $\mathsf{co\text{-}CDH}$ algorithm $\mathcal{A}$ as follows. On input $(\langle \mathsf{group} \rangle, A = G_1^{\alpha}, B_1 = G_1^{\beta}, B_2 = G_2^{\beta})$, algorithm $\mathcal{A}$ samples $r \xleftarrow{\$} \mathbb{F}$ and $\mathsf{pk}^* \leftarrow B_2, \pi^* \leftarrow B_1^r$, and $\mathsf{H_p}(B_2) \leftarrow G_1^r$. Next, $\mathcal{A}$ samples $k \in \{1, \ldots, q_{\mathsf{S}} + q_{\mathsf{H_s}}\}$ and runs $\mathcal{F}$ on input $(\mathsf{pp}, \mathsf{pk}^*, \pi^*)$ simulating its oracles as follows:

- $\mathsf{H_s}(m)$: if this is the k-th query to this oracle, add $(m, \perp)$ to L_{s} and return A. Else, sample $\rho \xleftarrow{\$} \mathbb{F}$, add (m, ρ) to L_{s}, and return G_1^{ρ}.
- $\mathsf{H_p}(\mathsf{pk})$: sample $\xi \xleftarrow{\$} \mathbb{F}$, add (m, ξ) to L_{p}, and return A^{ξ}.
- $\mathsf{Sign}(m)$: simulate an internal query $\mathsf{H_s}(m)$ and lookup m in L_{s}. If the corresponding $\rho = \perp$ abort, else return B_1^{ρ}.

If $\mathcal{A}$ doesn't abort and $\mathcal{F}$ succeeds, then with probability $\frac{1}{q_{\mathsf{S}}+q_{\mathsf{H_s}}}$ we have $\mathsf{H_s}(m^*) = A$. If this is not the case, $\mathcal{A}$ aborts. Otherwise, it holds that

$$\Sigma = A^{\sum_{j \in \mathcal{PK}^*} (\log \mathcal{PK}_j^*) + \sum_{\substack{i \in I^*, \\ j \in \mathcal{PK}_i}} \log \mathcal{PK}_{i,j}} \cdot \prod_{i \in [n] \setminus I^*} h_i^{\sum_{j \in \mathcal{PK}_i} \log \mathcal{PK}_{i,j}} \ ,$$

where $I^* \subseteq [n]$ is the list of indices for which $m_i = m^*$. Then $h_i = G_1^{\rho_i}$ for some $\rho_i \in L_{\mathsf{s}}$.

Observe that for each (pk, π) pair that $\pi = A^{\xi \log \mathsf{pk}}$ for some $\xi \in L_{\mathsf{p}}$, so $\mathcal{A}$ can compute $\pi^{\xi^{-1}} = A^{\log \mathsf{pk}}$. For every pk, $\mathcal{A}$ can also query $\psi(\mathsf{pk}) = G_1^{\log \mathsf{pk}}$. Hence, $\mathcal{A}$ can compute the ψ-$\mathsf{co\text{-}CDH}$ solution

$$\Sigma \cdot \prod_{\substack{j \in \mathcal{PK}^*, \\ j \neq \mathsf{pk}^*}} \pi_j^{-\xi_j^{-1}} \cdot \prod_{\substack{i \in I^*, \\ j \in \mathcal{PK}_i}} \pi_j^{-\xi_{i,j}^{-1}} \cdot \prod_{\substack{i \in [n] \setminus I^*, \\ j \in \mathcal{PK}_i}} \psi(\mathsf{pk}_{i,j})^{-\rho_i} \ .$$

□

Proof of Theorem 4.

Proof. As noted in Appendix B.5 we simplified our exposition in Section 5.2 by considering CRH a collision-resistant hash—treating it as if were already picked from a CRH family by sampling and fixing a key. We now consider a CRH family $\mathsf{CRH} : \mathcal{K} \times \{0,1\}^* \to \mathcal{B}$, as well as an injective encoding $\mathsf{Encode} : \mathcal{B} \to \{0,1\}^{b-t}$. We show that given an efficient distinguisher $\mathcal{D}$ that makes at most $Q(\lambda)$ queries such that

$$\left| \Pr\left[\mathcal{D}^{\mathcal{O}(\cdot)}(1^\lambda) = 1\right] - \Pr\left[k \leftarrow \mathcal{K} \;\middle|\; \mathcal{D}^{\mathsf{H}(k,\cdot)}(1^\lambda) = 1\right]\right| = \mu(\lambda) > \mathrm{negl}(\lambda) \ ,$$

where $\mathsf{H}(k, \cdot)$ is built with $\mathsf{CRH}(k, \cdot)$, Encode, and a RO $\mathcal{O}' : \{0,1\}^b \to \{0,1\}^c$ as in Construction 3, and where $\mathcal{O} : \{0,1\}^* \to \{0,1\}^\ell$ is a RO, we can build an adversary $\mathcal{A}$ that breaks the collision-resistance of CRH.

On input $k \leftarrow \mathcal{K}$, $\mathcal{A}$ runs $\mathcal{D}$, simulating its oracle by running $\mathsf{H}(k, \cdot)$ on its queries. Let $\mathsf{qt} = \{q_i, \mathsf{H}(k, q_i)\}_{i \in [Q]}$ be the query transcript between $\mathcal{D}$ and $\mathcal{A}$ (where wlog we assume queries unique). Let $\{h'_i\}_{i \in [Q]} = \{\mathsf{CRH}(k, q_i)\}_{i \in [Q]}$ be the intermediate hash values computed by $\mathcal{A}$ while simulating $\mathcal{D}$'s oracle. Let coll be the event that for $i \neq j$ there exists $h'_i = h'_j$. If coll happens, then $\mathcal{A}$ outputs the first colliding query pair (q_i, q_j); else $\mathcal{A}$ outputs two random messages.

Since $\mathcal{O}$ and $\mathcal{O}'$ are ROs and Encode is injective, it follows when $\neg\mathsf{coll}$ that the distributions qt and $\{q_i, \mathcal{O}(q_i)\}_{i \in [Q]}$ are identical. Therefore, event coll (coinciding with $\mathcal{A}$'s success) must happen with probability at least $\mu(\lambda) > \mathrm{negl}(\lambda)$. Since $\mathcal{D}$ runs in time $\mathrm{poly}(\lambda)$, it is easy to see $\mathcal{A}$ does as well. □

E Groth16 is an O-SNARK

E.1 O-SNARKs Overview

Introduced in [FN16] and defined in Appendix B.5, an O-SNARK for an oracle family $\mathbb{O}$ is, informally, a SNARK which provides knowledge soundness against adversaries having access to a uniformly sampled oracle $\mathcal{O} \xleftarrow{\$} \mathbb{O}$. This is particularly useful in modeling the case where a SNARK is proven over authenticated data such as cryptographic signatures, as done in our Plumo construction detailed in Section 4.

In [FN16] the authors provide several results of tangential interest to our setting. Their impossibility result shows that O-SNARKs do not exist with respect to every family of oracles (in the standard model, assuming the existence of one way functions). This may explain why few formally-proven general results about SNARKs over authenticated data have been published to date. The first constructive result shows how to guarantee O-SNARK security for signature scheme oracles in the hash-then-sign paradigm; however, this requires hashing an "extra" uniformly sampled λ bits where λ is the security parameter. Their second constructive result shows O-SNARK security for every signing oracle family under a some conditions; this considers only adversaries that query nearly the entire message space of the signature algorithm. Neither constructive result is likely to apply in practice without modifying either existing hash implementations or their usage, suggesting the need for O-SNARK proofs tailored to individual proof systems.

In this section we prove O-SNARK security for Groth16 in the algebraic group model, defined in Appendix B.5, with respect to any oracle family $\mathbb{O}$ for which q-DLog is $\mathbb{O}$-hard (as defined in in Appendix B.4). We then conclude that Groth16 is an O-SNARK with respect to our BBSGLRY signing oracle, under the reasonable assumption that q-DLog is $\mathbb{O}$-hard where $\mathbb{O}$ is the BBSGLRY oracle family. We assume familiarity with the Groth16 SNARK, a full description of which, including the full setup, prover, and verifier routines, can be found in [Gro16].

E.2 Other Groth16 Proofs

Multiple proofs of the security of Groth16 exist in the literature. The original author of the SNARK proves security in the generic group model in [Gro16], first demonstrating the existence of an extractor for the underlying NILP evaluated "in the exponent" to produce the SNARK. The authors of [BGM17] prove security in the GGM of Groth16 using a modified reduced-depth CRS more amenable to their trusted setup protocol. The authors of [FKL18] extend the result of [Gro16] by proving knowledge soundness in the algebraic group model (AGM), by relying on the pre-existence of an extractor for the Groth16 NILP, shown in [Gro16]. The AGM is a stronger model is preferable to the GGM in modeling security guarantees, since AGM gives adversaries more power and thus is closer to the real world (note that security under AGM implies security under GGM [FKL18]).

A useful but more complex and specialized effort at proving Groth16 secure was made by [Koh+21], in which the authors prove security of Groth16 in the AGM together with the trusted setup protocol of [BGG17], analysed as a unified protocol. Further, work on simulation extractability by [Bag+20] contains another AGM proof of Groth16, which relies on a lemma based on the analysis of [FKL18] for which only a proof sketch is given.

In this section we prove security based on the proof for the polynomial commitment of [GWC20], yielding the first formal proof that Groth16 is an O-SNARK for some oracle family. Other approaches mentioned above could also be modified to prove O-SNARK security. Indeed, the lemma used in [Bag+20] is similar to the one we use here from [GWC20], with our knowledge soundness proof being substantially similar to that of [Bag+20], differing primarily in presentation, and in our consideration of O-SNARK security.

Finally, we note that we prove knowledge soundness for the original Groth16 CRS and protocol, and not the reduced-depth CRS variant proposed by [BGM17] which Plumo uses in production and which other projects such as Zcash [Hop+21] have used. However, our proof is easily adaptable to the latter case, in which one can simply remove the details we take from the [Gro16] NILP proof and replace them with the corresponding arguments in [BGM17], which likewise analyze the verification equation of their protocol as a polynomial equality check with CRS trapdoor elements considered as indeterminants.

E.3 Groth16 as an O-SNARK

Following [GWC20], we denote a *real pairing check* in a security game with algebraic adversary $\mathcal{A}_{\mathsf{alg}}$ as any check of the form

$$(a_1 \cdot T_1) \cdot (T_2 \cdot a_2) = 0$$

where T_1, T_2 are matrices over a given field $\mathbb{F}$, and a_1, a_2 are the vectors of $\mathbb{F}$-elements whose encodings in $\mathbb{G}_1, \mathbb{G}_2$ respectively are output by $\mathcal{A}_{\mathsf{alg}}$.

Next, for $i \in \{1, 2\}$, let σ_i denote the list of elements in the reference string in group i, and let $N_i = poly(\lambda)$ denote the number of oracle queries made by $\mathcal{A}_{\mathsf{alg}}$. We define an *ideal pairing check* as any check which can be written as

$$(R_1 \cdot T_1) \cdot (T_2 \cdot R_2) \equiv 0$$

where each T_i is as above. R_i is the vector of polynomials

$$R_{i,j}(X) := \sum_{\ell=1}^{|\sigma_i|} v_\ell f_{i,\ell}(X) + \sum_{\ell'=1}^{N_i} v_{\ell'} f'_{i,\ell'}(X)$$

where $f_{i,j}$ is the polynomial for the jth element of σ_i, $f'_{i,j}$ is the polynomial corresponding to the jth oracle query response in group i, and the $v_\ell, v_{\ell'}$ are the elements an algebraic adversary outputs to explain its outputs as a linear combination of both its inputs and its observed oracle responses, i.e. $v_\ell, v_{\ell'}$ are such that $R_{i,j}(x) = a_{i,j}$.

As an example of an oracle response polynomial above, $f'(S, R) = R \cdot S$ would be the polynomial corresponding to a BLS signing query with secret key S and a discrete log R of the message hash relative to the generator of the group the signature is encoded in.

Our notion of ideal pairing check above is adapted from [GWC20] to the more general case of an algebraic O-SNARK adversary, which explains its output group elements not only in terms of its inputs, but also in terms of the responses it gets from its oracle. We similarly use a modified version of their Lemma 2.2 and proof suitable for our setting:

Lemma 1. *Given some oracle family $\mathbb{O}$, assume the $\mathbb{O}$-hardness of q-DLog. Given in addition an algebraic O-SNARK adversary $\mathcal{A}_{\mathsf{alg}}$ for $\mathbb{O}$, participating in a SNARK protocol with a degree Q reference string and a verification equation which can be written as a real pairing check, the probability with which this real pairing check passes is larger by at most an additional $negl(\lambda)$ factor over the probability that the corresponding ideal check passes.*

Proof. We construct an algebraic O-SNARK adversary $\mathcal{B}_{\mathsf{alg}}$ which runs $\mathcal{A}_{\mathsf{alg}}$, and breaks q-DLog with the probability of the difference between the probability that verification as a real pairing check passes and its corresponding ideal check passes.

$\mathcal{B}_{\mathsf{alg}}$ receives a q-DLog challenge with x as its base, then runs $\mathcal{A}_{\mathsf{alg}}$, simulating the verifier role of the protocol, and the oracle of $\mathcal{A}_{\mathsf{alg}}$ by querying its own oracle with the requested input of $\mathcal{A}_{\mathsf{alg}}$ and returning the result. $\mathcal{B}_{\mathsf{alg}}$ uses the vector of coefficients received from $\mathcal{A}_{\mathsf{alg}}$ to check if the real check passes and the ideal check failed, so that

$$(a_1 \cdot T_1) \cdot (T_2 \cdot a_2) = 0$$

and

$$(R_1 \cdot T_1) \cdot (T_2 \cdot R_2) \not\equiv 0$$

Otherwise, $\mathcal{B}_{\mathsf{alg}}$ aborts. $\mathcal{B}_{\mathsf{alg}}$ then computes the polynomial

$$(R_1 \cdot T_1) \cdot (T_2 \cdot R_2)$$

which has degree at most $2Q$ and x as a root, and so can be factored to find x. □

Next we state an important corollary, which is essentially a restatement of the Algebraic Verification Satisfiability lemma used for the Groth16 AGM proof in [Bag+20], and which is an equivalent formulation of Lemma 1 above:

Corollary 1. *Given some oracle family $\mathbb{O}$, assume the $\mathbb{O}$-hardness of q-DLog. Given an algebraic O-SNARK adversary $\mathcal{A}_{\mathsf{alg}}$ for oracle family $\mathbb{O}$, participating in a protocol with a degree Q reference string, the probability that both some real pairing check passes and the corresponding ideal pairing check fails is $negl(\lambda)$.*

Proof. Let $\mathcal{R}$ denote the event that some real pairing check passes, and $\mathcal{I}$ the event that the corresponding ideal pairing check passes. By construction we have that

$$\Pr[\mathcal{R}] = \Pr[\mathcal{I}] + \Pr[\mathcal{R} \wedge \neg\mathcal{I}]$$

While by Lemma 1:

$$\Pr[\mathcal{R}] = \Pr[\mathcal{I}] + \mathrm{negl}(\lambda)$$

□

Theorem E1. *Given an oracle family $\mathbb{O}$, the* Groth16 *SNARK is an O-SNARK with respect to $\mathbb{O}$ in the algebraic group model, satisfying perfect completeness, succinctness, and adaptive argument of knowledge assuming the $\mathbb{O}$-hardness of q-DLog.*

Proof. The proofs for perfect completeness and succinctness are identical across both the O-SNARK and SNARK settings, and in particular follow from the arguments in [Gro16].

Next we prove the adaptive argument of knowledge property against an algebraic O-SNARK adversary $\mathcal{A}_{\mathsf{alg}}$ with access to a uniformly sampled oracle $\mathcal{O}$ from $\mathbb{O}$. Let $S(\boldsymbol{X})$ be the polynomial corresponding to the Groth16 verification equaton, i.e.

$$S(\boldsymbol{X}) := -A \cdot B + \alpha \cdot \beta + \frac{\sum_{i=0}^{\ell} a_i(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau))}{\gamma} \cdot \gamma + C \cdot \delta$$

where $\boldsymbol{X}$ contains $(\alpha, \beta, \delta, \gamma, \tau)$. Note that, in the Groth16 argument of knowledge game, we have an ideal pairing check of the form

$$S(\boldsymbol{X}) \equiv 0$$

where $\boldsymbol{X}$ contains both $(\alpha, \beta, \delta, \gamma, \tau)$ as well as all of the oracle responses received by $\mathcal{A}_{\mathsf{alg}}$, while A, B, C are also understood as polynomials in these indeterminants as linear combinations of $\mathcal{A}_{\mathsf{alg}}$'s inputs and oracle responses.

We also have the corresponding real pairing check

$$S(\boldsymbol{x}) = 0$$

where A, B, C are the proof elements output by $\mathcal{A}_{\mathsf{alg}}$ and $\boldsymbol{x} = (\alpha, \beta, \delta, \gamma, \tau)$.

For our argument of knowledge adversary $\mathcal{A}_{\mathsf{alg}}$ to win, the real pairing check $S(\boldsymbol{x}) = 0$ must pass for its output $\pi = (A, B, C)$. There are two cases where this can occur. One in which the ideal pairing check $S(\boldsymbol{X}) \equiv 0$ passes, and another in which it does not pass. By Corollary 1, the latter case occurs with negligible probability.

Next we show the existence of an extractor in the case where the ideal pairing check passes. Let $q_1, ..., q_N$ denote polynomials in arbitrary oracle-specific indeterminants corresponding to the results of $\mathcal{A}_{\mathsf{alg}}$'s $N = poly(\lambda)$ queries to $\mathcal{O}$. We know that

$$A = A_\alpha \alpha + A_\beta \beta + A_\gamma \gamma + A(x) + \sum_{i=0}^{\ell} A_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} +$$
$$\sum_{i=\ell+1}^{m} A_i \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} + A_h(x)\frac{t(x)}{\delta} + A_{q_1} q_1 + ... + A_{q_N} q_N$$

where B and C can be written similarly, and each A_i, B_i, C_i can be found in the list of vectors output by $\mathcal{A}_{\mathsf{alg}}$ along with π as an algebraic adversary.

We can as in [Gro16] cancel out terms to obtain

$$A = \alpha + A(x) + A_\delta \delta + A_{q_1} q_1 + ... + A_{q_N} q_N$$

and similarly for B. We here note that in the verification equation

$$A \cdot B = \alpha \cdot \beta + \frac{\sum_{i=0}^{\ell} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma} \cdot \gamma + C \cdot \delta$$

we have $A_{q_i} B_{q_i} q_i^2 = 0$ and $\alpha q_i B_{q_i} = 0$ for all $1 \leq i \leq N$. Without loss of generality we can then assume $B_{q_i} = 0$ for all $0 \leq i \leq N$, implying also that all $A_{q_i} = 0$. Letting $a_i := C_i$ for $\ell + 1 \leq i \leq m$ we can then conclude that

$$\sum_{i=0}^{m} a_i u_i(x) \cdot \sum_{i=0}^{m} a_i v_i(x) = \sum_{i=0}^{m} a_i w_i(x) + C_h(x)t(x)$$

showing $(C_{\ell+1}, ..., C_m)$ is a witness.

□

Corollary 2. Groth16 *is an O-SNARK in the algebraic group model for the* BBSGLRY *oracle family* $\mathbb{O}$*, assuming the* $\mathbb{O}$*-hardness of q-DLog.*

Proof. This follows directly from Theorem E1. □

F The Plumo specification

We present a procedural description of our main circuit that implements our summary relation over BW6-761 described in Section 4.3. Here we take n to be the number of validators and N the number of epochs to be proved, where $N > 1$. For simplicity we will not cover epoch padding if the number of epochs to be proven is less than N, although we note that it is simple in practice to hard-code trivially satisfying the relevant circuit logic for dummy epochs.

We assume here the existence of a bilinear group, presented according to the notation in Appendix B.4. We also assume the circuit is implemented over a field $\mathbb{F}$.

When not dealing with subroutines, we use the notation Circuit$(x : w)$ to indicate that Circuit implements an NP-relation with public inputs x and private inputs w. We will also denote by $\mathbf{b}$ a bitmap of length n.

Main circuit

We first define the following helper methods:

- $\mathsf{EncodeEpochToBits}(i, r, \delta, \delta', t, apk, \{pk_i\}_{i=1}^n)$:
  1. Encode i, the epoch index, as a 16-bit integer.
  2. Encode r, the consensus round number, as an 8-bit integer.
  3. Encode t, the maximum number of non-signers, as a 32-bit integer.
  4. Encode δ, the current epoch entropy, in 128 bits.
  5. Encode δ', the parent epoch entropy, in 128 bits.
  6. Encode each public key in $\{pk_i\}_{i=1}^n$ as a $\mathbb{G}_2$ compressed point. If there are fewer public keys than the maximum defined in the system parameters, pad with G_2 until the maximum number of public keys is reached.

- $\mathsf{EncodeEpochToBitsEdges}(i, \delta, \delta', t, apk, \{pk_i\}_{i=1}^n)$:
  1. Encode i, the epoch index, as a 16-bit integer.
  2. If this is the first epoch, encode δ', the parent epoch entropy in 128 bits. If this is the last epoch, encode δ, the current epoch entropy in 128 bits.
  3. Encode t, the required signer threshold, as a 32-bit integer.
  4. If this is the last epoch, encode apk, the aggregated public key of this validator set, as a compressed $\mathbb{G}_2$ point.
  5. Encode each public key in $\{pk_i\}_{i=1}^n$ as a $\mathbb{G}_2$ compressed point. If there are fewer public keys than the maximum defined in the system parameters, pad with G_2 until the maximum number of public keys is reached.

  Next we describe the main circuit. In the following let

$$E_j = \{i_j, r_j, \delta_j, \delta'_j, t_j, apk_j, \{pk_{j,k}\}_{k=1}^n\}$$

A subroutine taking as input some E_j is assumed to discard those elements included in it which are not a part of the subroutine's input.

- $\mathsf{MainCircuit}(H'(e_1), H'(e_N) : \sigma_{agg}, \{H(e_j)\}_{j=2}^N, \{\mathbf{b}_j\}_{j=1}^{N-1}, \{E_j\}_{j=1}^N)$:
  1. For each $j = 2...N$ perform:
     (a) Check that $apk_{j-1} =? \sum_{i=1}^n b_i \cdot pk_{j-1,i}$ where b_i is the i-th bit of $\mathbf{b}_{j-1}$.
     (b) Check that $\delta_{j-1} =? \delta'_j$
     (c) Check that $i_{j-1} =? i_j + 1$
     (d) Encode E_j as e_j using $\mathsf{EncodeEpochToBits}$ and hash it using $\mathsf{BHPedersenHash}$. Then, run $\mathsf{Blake2Xs}$ on the intermediate result to obtain the final result of the composite hash. Finally, complete the hash following the hash-to-group method described in Sections 5 and 6.1. Check that the result is equal to $H(e_j)$.
  2. Check that $apk_N =? \sum_{i=1}^n pk_{N,i}$.
  3. Check that $e(\sigma_{agg}, G_2^{-1}) \cdot e(H(e_2), apk_1) \cdot ... \cdot e(H(e_n), apk_{n-1}) =? 1_{\mathbb{G}_T}$
  4. Encode E_1 as e_1 and E_N as e_N each using $\mathsf{EncodeEpochToBitsEdges}$. Hash individually both e_1 and e_N directly with $\mathsf{Blake2s}$. Tightly pack, individually, the first and last epoch resulting hash bits into elements of $\mathbb{F}$. Check that the results of this packing are equal to $H'(e_1), H'(e_N)$ respectively.

References

[Al-+18] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. "Chainspace: A Sharded Smart Contracts Platform". In: *Procedings of the 25th Network and Distributed System Security Symposium*. NDSS '18. 2018 (3).

[Alb+16] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. "MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity". In: *22nd International Conference on the Theory and Application of Cryptology and Information Security*. 2016, pp. 191–219 (7).

[Amo+18] Y. Amoussou-Guenou, A. Del Pozzo, M. Potop-Butucaru, and S. Tucci Piergiovanni. "Correctness of Tendermint-Core Blockchains". In: *22nd International Conference on Principles of Distributed Systems*. Vol. 125. OPODIS '18. 2018, 16:1–16:16 (3, 16).

[Aum+13] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. "BLAKE2: simpler, smaller, fast as MD5". In: *11th International Conference of Applied Cryptography and Security*. ACNS '13. 2013 (7, 12, 13).

[AVL62] G. Adelson-Velsky and E. Landis. "An algorithm for the organization of information". In: *USSR Academy of Sciences*. 1962 (16).

[Bag+20] K. Baghery, M. Kohlweiss, J. Siim, and M. Volkhov. *Another Look at Extraction and Randomization of Groth's zk-SNARK*. Cryptology ePrint Archive, Report 2020/811. 2020 (27, 29).

[BDN18] D. Boneh, M. Drijvers, and G. Neven. "Compact Multi-signatures for Smaller Blockchains". In: *24th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '18. 2018, pp. 435–464 (5, 7, 17, 20–22).

[Ben+14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves". In: *34th Annual International Cryptology Conference*. CRYPTO '14. 2014, pp. 276–294 (4).

[BGG17] S. Bowe, A. Gabizon, and M. Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. Cryptology ePrint Archive, Report 2017/602. 2017 (27).

[BGH19] S. Bowe, J. Grigg, and D. Hopwood. *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Report 2019/1021. 2019 (4).

[BGM17] S. Bowe, A. Gabizon, and I. Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Report 2017/1050. 2017 (24, 27, 28).

[BGR98] M. Bellare, J. A. Garay, and T. Rabin. "Fast Batch Verification for Modular Exponentiation and Digital Signatures". In: *17th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '98. 1998, pp. 236–250 (14, 25).

[Bit+13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data". In: *45th ACM Symposium on the Theory of Computing*. STOC '13. 2013, pp. 111–120 (11).

[Bit+16] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen. "On the Existence of Extractable One-Way Functions". In: *SIAM Journal on Computing* 45.5 (2016). Preliminary version appeared in STOC '14., pp. 1910–1952 (11).

[BLS01] D. Boneh, B. Lynn, and H. Shacham. "Short Signatures from the Weil Pairing". In: *7th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '01. 2001, pp. 514–532 (7, 13, 17).

[Bol03] A. Boldyreva. "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme". In: *6th International Conference on Practice and Theory in Public Key Cryptography*. PKC '03. 2003, pp. 31–46 (7, 17).

[Bon+03] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *22nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '03. 2003, pp. 416–432 (7, 17).

[Bon+20] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro. *Coda: Decentralized Cryptocurrency at Scale*. Cryptology ePrint Archive, Report 2020/352. 2020 (3, 4).

[Bow+20] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. "Zexe: Enabling Decentralized Private Computation". In: *41st IEEE Symposium on Security and Privacy*. S&P '20, 2020, pp. 947–964 (5, 8).

[Bra+20] S. Braithwaite et al. "A Tendermint Light Client". In: (2020) (16).

[BSCS16] E. Ben-Sasson, A. Ciesa, and N. Spooner. "Interactive Oracle Proofs". In: *14th Theory of Cryptography Conference*. TCC '16. 2016 (4).

[Bün+18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *39th IEEE Symposium on Security and Privacy*. S&P '18. 2018, pp. 315–334 (16).

[Bün+20a] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. "Recursive Proof Composition from Accumulation Schemes". In: *18th Theory of Cryptography Conference*. Vol. 2. TCC '20. 2020, pp. 1–18 (4).

[Bün+20b] B. Bünz, L. Kiffer, L. Luu, and M. Zamani. "FlyClient: Super-Light Clients for Cryptocurrencies". In: *41st IEEE Symposium on Security and Privacy*. S&P '20. 2020, pp. 928–946 (3, 10, 18).

[CCW19] A. Chiesa, L. Chua, and M. Weidner. "On Cycles of Pairing-Friendly Elliptic Curves". In: *SIAM Journal on Applied Algebra and Geometry* 3.2 (2019), pp. 175–192 (4).

[CGR11] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd ed. Springer, 2011 (19).

[Che10] J. H. Cheon. "Discrete Logarithm Problems with Auxiliary Inputs". In: *Journal of Cryptology* 23.3 (2010), pp. 457–476 (25).

[Che+20] W. Chen, A. Chiesa, E. Dauterman, and N. P. Ward. *Reducing Participation Costs via Incremental Verification for Ledger Systems*. Cryptology ePrint Archive, Report 2020/1522. 2020 (3–5).

[Chi+20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward. "Marlin: Preprocessing zkSNARKS with Universal and Updatable SRS". In: *39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '20. 2020, pp. 738–768 (4, 21).

[CKK20] S. Cao, S. Kadhe, and R. Kannan. *CoVer: Collaborative Light-Node-Only Verification and Data Availability for Blockchains*. ArXiv abs/2010.07031. 2020 (16).

[CL06] J. H. Cheon and D. H. Lee. "Use of Sparse and/or Complex Exponents in Batch Verification of Exponentiations". In: *IEEE Transactions on Computers* 55.12 (2006), pp. 1536–1542 (14).

[CT10] A. Chiesa and E. Tromer. "Proof-Carrying Data and Hearsay Arguments from Signature Cards". In: *1st Conference on Innovations in Computer Science*. ICS '10. 2010, pp. 310–331 (4).

[EHG20] Y. El Housni and A. Guillevic. *Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition*. Cryptology ePrint Archive, Report 2020/351. 2020 (8).

[FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss. "The Algebraic Group Model and its Applications". In: *38th Annual International Cryptology Conference*. CRYPTO '18. 2018, pp. 33–62 (11, 21, 27).

[FN16] D. Fiore and A. Nitulescu. "On the (In)Security of SNARKs in the Presence of Oracles". In: *14th International Conference on the Theory of Cryptography*. TCC '16. 2016, pp. 108–138 (9, 11, 23, 27).

[GPS08] S. D. Galbraith, K. G. Paterson, and N. P. Smart. "Pairings for cryptographers". In: *Discrete Applied Mathematics* 156.16 (2008), pp. 3113–3121 (20).

[Gra+19] L. Grassi, D. Kales, D. Khovratovich, A. Roy, C. Rechberger, and M. Schofnegger. "Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems". In: (2019) (5, 7).

[Gro16] J. Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *35th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT '16. 2016, pp. 305–326 (27–30).

[Gud+19] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. *SoK: Off The Chain Transactions*. Cryptology ePrint Archive, Report 2019/360. 2019 (3).

[GWC20] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2020/315. 2020 (27, 28).

[Hop+21] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. *Zcash Protocol Specification [Overwinter+Sapling]*. 2021 (7, 12–14, 28).

[KK+16] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *25th USENIX Conference on Security Symposium*. USENIX Security '16. 2016, pp. 279–296 (4).

[KK+18] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *39th IEEE Symposium on Security and Privacy*. S&P '18. 2018, pp. 583–598 (3).

[KMZ20] A. Kiayias, A. Miller, and D. Zindros. "Non-interactive Proofs of Proof-of-Work". In: *24th International Conference on Financial Cryptography and Data Security*. FC '20. 2020, pp. 505–522 (3).

[Koh+21] M. Kohlweiss, M. Maller, J. Siim, and M. Volkhov. *Snarky Ceremonies*. Cryptology ePrint Archive, Report 2021/219. 2021 (24, 25, 27).

[Mal+17] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. "Concurrency and Privacy with Payment-Channel Networks". In: *2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Association for Computing Machinery, 2017, pp. 455–471 (3).

[Mal+19] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. "Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings". In: *26th ACM Conference on Computer and Communications Security*. CCS '19. 2019, pp. 2111–2128 (4).

[Mei18] S. Meiklejohn. "Top Ten Obstacles along Distributed Ledgers Path to Adoption". In: *IEEE Security and Privacy* 16.4 (2018), pp. 13–19 (3).

[Mon20] H. Moniz. *The Istanbul BFT Consensus Algorithm*. ArXiv abs/2002.03613. 2020 (6, 11, 19).

[Nik+17] K. Nikitin et al. "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds". In: *26th USENIX Security Symposium*. USENIX Security '14. 2017, pp. 1271–1287 (3).

[PB17] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts*. 2017 (16).

[RY07] T. Ristenpart and S. Yilek. "The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks". In: *26th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '07. 2007, pp. 228–245 (7, 17, 22).

[Yin+19] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *ACM Symposium on Principles of Distributed Computing 2019*. PODC '19. 2019, pp. 347–356 (3).

[Zam+20] A. Zamyatin, Z. Avarikioti, D. Perez, and W. J. Knottenbelt. "TxChain: Efficient Cryptocurrency Light Clients via Contingent Transaction Aggregation". In: *4th International Workshop on Cryptocurrencies and Blockchain Technology*. CBT '20. 2020, pp. 269–286 (16).

[Zha+20] W. Zhang, J. Yu, Q. He, N. Zhang, and N. Guan. "TICK: Tiny Client for Blockchains". In: *IEEE Internet of Things Journal*. IOT-J '20 (2020) (16).