# Plumo: Towards Scalable Interoperable Blockchains Using Ultra Light Validation Systems

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

[1]AZTEC Protocol
`ariel@aztecprotocol.com`

[2]cLabs
`{a,kobi,m,mstraka,psi}@c-labs.co`

[3]Independent Researcher
`me@gakonst.com`

[4]Tel Aviv University
`tromer@cs.tau.ac.il`

[5]University College London
`p.jovanovic@ucl.ac.uk`

[6]Columbia University, [7]Ethereum Foundation, [8]UC Berkeley

April 1, 2020

**Abstract**

Scalability and interoperability issues have been two of the main reasons preventing the wide-spread adoption of blockchain systems. Despite significant progress in recent years, solutions that are simple, efficient, and secure, and that do not restrict functionality remain elusive. In this paper, we are taking a step forward to address these challenges by introducing Plumo, a framework that enables secure and efficient light client synchronization and cross-chain transaction validation via SNARK proofs.

We present a formal framework for Plumo, as well as a concrete instantiation on top of a BFT consensus network which uses SNARKs to prove changes in the consensus committee. We suggest optimized building blocks, including SNARK-friendly hash-to-curve functions from existing cryptographic primitives and SNARK-friendly aggregateable BLS signatures.

Finally, we present an evaluation of our implementation, showing that even resource-constrained clients (such as low-end mobile phones) can transact efficiently and securely with our system. We show that the SNARK proof spanning 6 months worth of blockchain validator updates are quite feasible: proving costs about USD $12 worth of computation on modern cloud infrastructure, and verification can be done in about 6 seconds on a low-end smart phone.

[*]Corresponding author.

## 1 Introduction

In recent years, there have been significant advancements to address crucial scalability and interoperability challenges in the blockchain space. On the scalability side this includes the development of more efficient consensus protocols to improve latency and throughput [BKM18; KK+16; Yin+19], sharding to allow for better utilization of available network resources further improving transaction processing throughput [Al-+18; Cor+13; KK+], and payment channels to reduce the amount of transaction data that needs to be processed and stored on-chain while also providing better performance [Gud+; Mal+17]. On the interoperability side improvements include new insights on atomic cross-chain swaps [EMSM19; Her18], symmetric cryptographic locks [Mal+20; TMSM19], (custodial) exchange protocols [Ben+19; HLG19], and other asset transfer techniques [KK+; TSB19; Zam+19]. Despite all of this progress, however, several key challenges still remain understudied.

In terms of scalability, efficient synchronization and verification of a distributed ledger or parts thereof are still a major hurdle for clients who want to bootstrap or catch up on the latest state, since they have to spend significant resources in terms of bandwidth and computation. For example, at the time of writing (January 2020), the Ethereum blockchain (in non-archive mode) has a size of about 250 GB, including more than 4.5 GB of header metadata[1]. Such heavy workloads are infeasible for all but the most resource-rich participants, who are able and willing to run full nodes. This leaves out a large potential user base, including (light) clients running on mobile phones, and clients who interact with the network only intermittently. This issue is expected to become even more severe with the current strive for low-latency, high-throughput chains to provide better usability which results in significant chain growth in relatively short time periods. First attempts to address this challenge have been proposed recently but they are arguably too complex and constrained for wide-spread use [MS18], limited to networks that are based on Proof-of-Work and Nakamoto consensus [Bün+19b], or require to keep (old) consensus committees active for extended periods of time to create on-chain forward links [Nik+17] which increases the risk of long-range blockchain forking attacks of forking the blockchain.

Another motivating use case is cross-chain interoperability [Zam+19], which nowadays are either limited in their functionality [Int] or rather complex and, *e.g.*, require entire arbiter chains [Cos; Pol]. In many cases, to cope with failures in cross-chain communication, the verifying chain has to retrieve and verify all headers of the source chain [Btc; Wze], which is (as argued above) unscalable.

**Our contribution.** We introduce the Plumo validation framework to enable lightweight transaction validation without requiring clients to download and verify the full header chain. The framework uses succinct non-interactive arguments of knowledge (SNARKs) [Bit+12; Gen+13; Gro16] to produce short proofs attesting to the validity of large amounts of blockchain data. These proofs, once published, can be cheaply verified by light clients or in cross-chain transaction validation.

We further present a concrete instantiation of our framework on top of a BFT consensus network [BKM18; KK+16; Yin+19] and describe our implementation and evaluation results. This allows even resource-constrained clients, such as low-end mobile phones, to transact efficiently and safely in modern high-performance blockchain networks, by using the SNARK proofs to verify changes in the set of authoritative blockchain validators. Besides presenting our optimized SNARK relations for BFT consensus committee transitions, we make several additional contributions to further improve efficiency of SNARK processing, including a generic approach to build SNARK-friendly hash-to-curve functions from established cryptographic primitives and present a concrete

[1]Calculated as 508 byte headers times the block number 9360646 Statistics from `https://etherscan.io/charts`.

instantiation combining Bowe-Hopwood hash [Hop+19] and Blake2Xs [Aum+16] resulting in SNARK-friendly BLS signing [BGS03; BLS01].

We implemented the BFT-based instantiation of the PLUMO validation framework, using the Groth16 proof system [Gro16] and SCIPR Lab implementation of Zexe [Bow+18], our SNARK-friendly hash-to-curve function [Hop+19] and BLS signing. We integrated this with the Celo blockchain. For our experiments we used a Google Cloud n1-ultramem machine[2] which is equipped with 4 Intel® Xeon® E7-8880 v4 processors and 3844 GB of DDR4 RAM. On this machine, it takes about 47 (86) min to create the SNARK proof covering 128 (256) of daily validator changes encoded in about 67 M (127 M) constraints. This effort is required just once per such multi-month interval, and the proof can then be verified by anyone. Independently of the number of validator changes, these proofs have a size of less than 1 kB and can be verified efficiently even on low-end phones such as the Motorola Moto G (2nd Gen) in about 6 seconds with an unoptimized implementation.

This paper makes the following contributions:

- We introduce the PLUMO validation framework for secure and more efficient light client synchronization and cross-chain transaction validation.
- We provide a formal security analysis for our PLUMO validation framework, which might be also of independent interest.
- We describe a generic approach to build SNARK-friendly hash-to-curve functions from established primitives together with a concrete instantiation designed to improve prover efficiency on large messages, SNARK-friendly BLS signing instantiation, and our SNARK relations to prove validator changes.
- We present details of our implementation and evaluation of a concrete instantiation of the PLUMO validation framework showing that its cheap to generate and verify SNARK-based compression proofs covering months worth of validator changes.

The rest of the paper is organized as follows: We discuss related work in Section 2, give a system overview in Section 3, introduce preliminaries in Section 4, describe the PLUMO validation framework in Section 5, the protocol instantiation in Section 7, and implementation and evaluation results in Sections 8 and 9, respectively. We conclude with future work in Section 10.

## 2 Related Work

Blockchain scalability has attracted a good amount of attention over the past few years, due to the desirability of and technical difficulties involved in creating a decentralized ledger which is easy to use. There are two key problems; obtaining good *throughput* of transactions processed, and ensuring fast *validation time* so that nodes transacting using the underlying protocol can be confident that the state of the ledger they have is correct, without performing an undue amount of computation.

Several approaches have been proposed so far which improve one or both of these two properties. We note here that we have excluded from discussion approaches which significantly trade off decentralization in favor of efficiency. Without such a cutoff we would necessarily include here a history of centralized databases, which is not our focus.

**BFT Consensus.** One throughput bottleneck in Bitcoin (and similar) blockchains is the need to broadcast all transaction information to all potential Proof of Work miners. Better scalability is

[2]https://cloud.google.com/blog/products/gcp/introducing-ultramem-google-compute-engine-machine-types

attained by having, at any given time, a small committee of well-resourced validators who are in charge of agreeing on new blocks, using a *Byzantine Fault Tolerance (BFT)* consensus protocol [CL99; BKM18; KK+16; Mon20; Yin+19]. In permissionless blockchains committee members create their identities via a Sybil-resistant identity creation mechanism such as *Proof of Work (PoW)* [Nak09; KK+16] or more commonly *Proof of Stake (PoS)* [Kia+17] and are elected into the committee through some randomized voting process [Bon+18; Gal+20; Syt+17]. We will not review the current state of consensus protocols in this paper, save to note that PLUMO's improvements apply on top of those offered by committee-based consensus.

**Scalability for Proof of Work Chains.** PoW chains, while typically lower in throughput, have so far seen more success in implementing scalability solutions which improve validation time for clients of the chain. One such project is NiPoPoW, which uses statistical properties of the proof of work hashes to make probabilistic guarantees about the validity of the chain in logarithmic time [KLS16]. FlyClient, a subsequent work, takes this concept further by effectively placing block hashes in a tree structure [Bün+19a]. These approaches, however, do not apply to proof of stake chains due to the lack of such statistical properties in the cryptographic signing algorithms they leverage.

**SNARK-based Scaling.** There are already a few approaches that use SNARK-based techniques to improve scalability of blockchain systems similar to PLUMO. Coda [Mec] maintains a succinct summary of the blockchain state whose correctness is attested to by SNARK proofs, which can be incrementally updated by recursive composition of SNARKs a la Proof-Carrying Data [Val08; Bit+13; Ben+14a]. The benefit of this approach is that the latest chain state can always be verified in a few milliseconds. One downside is efficiency of updates: they always require running the SNARK prover, and moreover, involve relatively slow elliptic curves. [3] Another downside is that all blockchain verification logic needs to be implemented as constraint system to be proven by the SNARK, which is difficult to implement and constrains functionality. Piperine [LNS20] focuses on reducing costs to prove and verify complex state machine transitions and shows how to use its SNARK-based scaling techniques for Ethereum.

**Layer Two Scaling.** Several other approaches for blockchain scalability have been proposed. In the common parlance of the industry, *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 rollup [Glu] is 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.

While some of the above approaches address the tension between the two outcomes of improved throughput and fast chain validation, they do so by introducing additional complications. Prior approaches using SNARKs require either verifying a large number of proofs (which is impractical for low-end devices), or very expensive prover costs (which neessitates new economic incentives, a

[3]The practical way to do repeated recursive compositions is via a cycle of MNT elliptic curves [Ben+14a]; due to their low embedding degrees (4 and 6), the base fields of these curves require nearly 800 bits to achieve 128 bits of security, which slows down both proving and verification. Recent results indicate difficulty of finding more efficient cycles of curves [CCW18].

presumption of generosity on the part of the community, or centralization in availability). Approaches not using SNARKs achieving both properties currently broaden their attack surface by introducing additional assumptions of economic rationality and the bounded external incentives, which may not hold in reality.

PLUMO thus seems to be the first to achieved high throughput and fast chain validation, without major new assumptions. Concretely, in the parameter chosen for our implementation, we merely need someone to generate a SNARK proof every 6 months at a cost of about \$12, and without increasing the protocol's attack surface beyond standard BFT+PoS and the use of well-studied SNARK schemes.

## 3 System Overview

We now describe the ideas and tradeoffs underlying the PLUMO system, exemplified by its concrete application to the Celo blockchain. Our primary goal, here, is *light client efficiency*: light clients should be able to securely synchronize to the head of the chain by downloading a small amount of data and perform a short computation. A related goal is that even after synchronization to the latest state, processing subsequent blocks should also be efficient.

### 3.1 System model

We assume a blockchain network where new blocks are chosen by BFT consensus executed by a set of validators, with some fixed target block time. We assume that at any time there are $n$ validators. The set of validations can change (e.g., based on Proof-of-Stake elections) at the end of every *epoch*, which is some fixed number of blocks. Each validator has a public key signing $pk_i$ (e.g., registered together with the proof-of-stake before elections occur), and signs blocks with the corresponding secret signing key.

We assume that a light client that would like to make efficient queries to the latest state only has to know the current head of the chain.

### 3.2 Threat model

We assume that within each epoch, out of the present $n$ validators, more than $\frac{2n}{3}$ are honest. The remaining $f$ validators are assumed to be $f$ malicious (Byzantine), i.e., controlled by a adversary. They can collude, send invalid messages during consensus, or fail to send messages. Thus, signatures by a quorum of $\geq \frac{2n}{3}$ validators is needed for a block to be considered valid.

Any validator may be corrupted in subsequent epochs, after it has exited the validators set (i.e., during the epoch, validators' honest behavior is incentivized by slashing of its locked stake, but the lock is later lifted).

We assume that the adversary is computationally bounded, and that the cryptographic building blocks are indeed secure by their own definition. Our SNARK scheme relies on a structured reference string, which we assume is securely generated by a suitable MPC setup ceremony (i.e., that at least one participant in this MPC behaves honestly and does not leak their random coins).

## 3.3 Designing an ultra light client

We now describe a high-level design of a Plumo client, exemplified by the the Celo use-case, in a series of didactic steps. The presentation here stresses the high-level intuition; see Section 7 for further details.

**Initial strawman design** We could take the approach mentioned in the Bitcoin whitepaper [Nak09] and follow the Simple Payment Verification approach: download the headers of all the blocks, and verify that a quorum of $\geq \frac{2n}{3}$ currently approved validators signed each block. Whenever there is an election, update the currently approved validator set. This approach certainly works, but is inadequately efficient for light clients: there are too many headers to download, especially when bock times are short (which is desirable for for low transaction latency and round-trip times).

**Epoch-based syncing** An improvement is to allow elections only at some block interval, a period which we call an epoch, e.g. once a day. Since we're using a BFT-style consensus algorithm, light clients only need to know the current validator set in order to validate the most recent block. Therefore, we allow light clients to synchronize only epoch-transition blocks until they reach the head of the chain. Since the validator set changes only at those blocks, a light client can verify the evolution of the validator set (and are trusting the validators to validate the content of blocks within the epoch). Additionally, by following Ethereum state design, it's enough to know the latest state root to make efficient light client queries.

**Signature aggregation** Every block needs to include enough validator signatures (a quorum) to convince a light client of its validity. Signature aggregation enables replacing all the signatures with a single signature. We use BLS signatures for signing individual block headers by $\geq \frac{2n}{3}$ validators. This allows us to use the non-interactive aggregation functionality of BLS signatures, having a single signature in a block instead of $\geq \frac{2n}{3}$ signatures. To achieve a threshold-like functionality, we use an $n$-bit bitmap indicating which of the $n$ validators contributed their signature to the aggregated multisignature. Since the public keys are known, this suffices for verification.

**SNARK proofs for many epochs** After introducing epoch-based syncing and signature aggregation, the computation becomes simple enough to be succinctly summarized into a SNARK proof. The proof attests that the validator set evolved, over some large number of epochs (each accompanied by suitable signatures by that epoch's validators set), from a given initial validators set $A$ to a given final validator set $B$. Such a SNARK proof can aggregate many epochs transitions into a single transition from $A$ to $B$, and is highly succinct: the light client verifying this proof does not need to see the intermediate blocks, headers or even validator sets. This architecture is illustrated in Figure 3.1.

Before proceeding to describe the in-depth instantiation, we discuss an abstract Validation Framework and provide proofs of its security in Section 5.

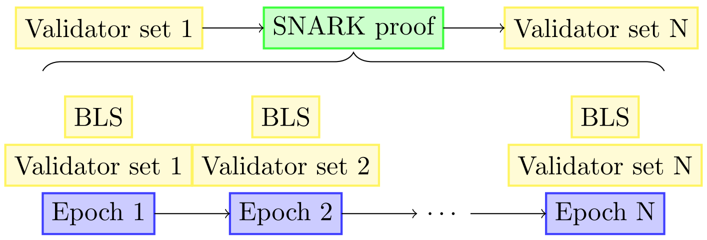

Figure 3.1: Plumo architecture overview

## 4 Preliminaries

### 4.1 Notation

We denote by $[n]$ the set $\{1, \ldots, n\} \subseteq \mathbb{N}$. 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 $\{a_i\}_1^n$ 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 $x \overset{\$}{\leftarrow} S$ denote that $x$ is an element sampled uniformly at random from $S$.

**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 $\lambda$ (namely, a function that vanishes faster than the inverse of any polynomial in $\lambda$). When a function can be expressed in the form $1 - \text{negl}(\lambda)$, we say that it is *overwhelming* in $\lambda$. 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 $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$.

### 4.2 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 $\lambda$ (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.

Generally, we distinguish between three types of bilinear group samplers [GPS08]: 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., $\mathsf{SXDH}$ only holds for Type III groups), and in general in this work we assume we are working with working with a Type III groups.

## 4.3 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 4.1** (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 4.2** (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.*

## 4.4 Aggregate multisignatures

Our definitions of an aggregate multisignature scheme follow [BDN18]. The only significant difference between our definitions stems from the fact we assume that a multisignature can be produced from a set of signatures offline by an untrusted party without secrets. Thus, instead of $\mathsf{Sign}$ taking a list of public keys in order to produce a multisignature, the multisignature functionality has been separated out into the $\mathsf{MultiSign}$ algorithm. An aggregate multisignature scheme consists of a 8-tuple of efficient algorithms ($\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{Verify}, \mathsf{AggSign}, \mathsf{VerifyAgg}$) that behave as follows:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$ : a public-coin setup algorithm that, given a security paramater $\lambda$ (represented in unary), outputs a set of public system parameters $\mathsf{pp}$.
- $\mathsf{KeyGen}(\mathsf{pp}) \xrightarrow{\$} (\mathsf{pk}, \mathsf{sk})$ : a key generation algorithm that, given public parameters $\mathsf{pp}$, outputs a public-secret key pair $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m) \to \sigma$ : a signing algorithm that, given a secret key $\mathsf{sk}$ and message $m \in \{0,1\}^*$, returns a signature $\sigma$.
- $\mathsf{KeyAgg}(\mathsf{pp}, \{\mathsf{pk}_i\}_{i=1}^n) \to \mathsf{apk}$ : an offline key aggregation algorithm that, given a set of $n$ public keys $\{\mathsf{pk}_i\}_{i=1}^n$, returns an aggregate public key $\mathsf{apk}$.
- $\mathsf{MultiSign}(\mathsf{pp}, [\sigma_i]_{i=1}^n) \to \sigma$ : an offline multisignature algorithm that, given $n$ signatures $[\sigma_i]_{i=1}^n$, returns a multisignature $\sigma$.
- $\mathsf{Verify}(\mathsf{pp}, \mathsf{apk}, m, \sigma) \to 0/1$ : a verification algorithm that, given aggregate public key $\mathsf{apk}$, message $m \in \{0,1\}^*$, and multisignature $\sigma$, returns 1 or 0 to indicate the multisignature is valid or invalid, respectively.
- $\mathsf{AggSign}(\mathsf{pp}, \{(\mathsf{apk}_i, m_i, \sigma_i)\}_{i=1}^n) \to \Sigma$ : an offline signature aggregation algorithm that, given a set of $n$ (aggregate public key, message, multisignature)-triplets $\{(\mathsf{apk}_i, m_i, \sigma_i)\}_{i=1}^n$, outputs aggregate multisignature $\Sigma$.
- $\mathsf{VerifyAgg}(\mathsf{pp}, \{(\mathsf{apk}_i, m_i)\}_{i=1}^n, \Sigma) \to 0/1$ : an aggregate multisignature verification algorithm that, given a set of $n$ (aggregate public key, message)-pairs $\{(\mathsf{apk}_i, m_i)\}_{i=1}^n$ and an aggregate multisignature $\Sigma$, returns 1 or 0 to indicate the signature is valid or invalid, respectively.

We require that an aggregate multisignature scheme ($\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{Verify}, \mathsf{AggSign}, \mathsf{VerifyAgg}$) satisfies completeness and unforgeability, as defined in 4.3.

**Definition 4.3** (Perfect completeness). *The aggregate multisignature scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{Verify}, \mathsf{AggSign}, \mathsf{VerifyAgg})$ *has perfect completeness if for all* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ *and* $\{\{((\mathsf{sk}_{i,j}, \mathsf{pk}_{i,j}), m_i)\}_{j=1}^{n_i}\}_{i=1}^{n}$, *where* $(\mathsf{sk}_{i,j}, \mathsf{pk}_{i,j}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$ *and the* $m_i$ *are distinct, it holds that*

$$\Pr\left[\mathsf{VerifyAgg}(\mathsf{pp}, \{(\mathsf{apk}_i, m_i)\}_{i=1}^{n}, \mathsf{AggSign}(\mathsf{pp}, \{(\mathsf{apk}_i, m_i, \sigma_i)\}_{i=1}^{n})) = 1\right] = 1 \ ,$$

*where for all* $i \in [n]$

$$(\mathsf{apk}_i, \sigma_i) \leftarrow \Big(\mathsf{KeyAgg}(\mathsf{pp}, \{\mathsf{pk}_{i,j}\}_{j=1}^{n_i}), \mathsf{MultiSign}(\mathsf{pp}, [\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}_{i,j}, m_i)]_{j=1}^{n_i})\Big) \ .$$

We note that an aggregate multisignature scheme with perfect completeness implies both a multisignature scheme and a signature scheme with perfect completeness.

Next, we define an unforgeable aggregate multisignature scheme. Informally, in the unforgeability game an adversary is given a challenge public key and signing oracle access to the corresponding secret key. Their goal is to output a valid aggregate multisignature for a set of (aggregate public key, message)-pairs, where one of the aggregate public keys can be formed by running KeyAgg on a set of public keys that includes the challenge key, and the corresponding message was not previously queried.

**Definition 4.4** (Computationally unforgeable aggregate multisignature). *For an aggregate multisignature scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{Verify}, \mathsf{AggSign}, \mathsf{VerifyAgg})$ *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.*

```
$\underline{\mathsf{Game}_{\mathcal{A}}^{\mathsf{forge}}(1^\lambda)}$
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\mathsf{pk}^*, \mathsf{sk}^*) \xleftarrow{\$} \mathsf{KeyGen}(\mathsf{pp})$
$Q \leftarrow \emptyset$
$(\{(\mathsf{apk}_i, m_i)\}_{i=1}^{n-1}, \Sigma, \{\mathsf{pk}_i\}_{i=1}^{t}, m_n) \leftarrow \mathcal{A}^{\mathsf{Sign}}(\mathsf{pp}, \mathsf{pk}^*)$
$\mathsf{apk}_n \leftarrow \mathsf{KeyAgg}(\mathsf{pp}, \{\mathsf{pk}_i\}_{i=1}^{t} \cup \mathsf{pk}^*)$
If $m_n \notin Q \ \wedge \ \mathsf{VerifyAgg}(\mathsf{pp}, \{(\mathsf{apk}_i, m_i)\}_{i=1}^{n}, \Sigma)$ :
   Return 1
Else return 0

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

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

Note that an unforgeable aggregate multisignature scheme implies both an unforgeable multisignature scheme and an unforgeable signature scheme as in the definitions given presented in [BDN18].

## 4.5 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. Our interface and definitions present a simplification of theirs by removing consideration of auxillary inputs and some relaxing some performance requirements including on Prove and Setup.

**Definition 4.5.** *A succinct non-interactive argument of knowledge for a relation $\mathcal{R}$ and an oracle family $\mathbb{O}$ is a triple of efficient algorithms $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ such that:*

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

*Additionally, $\Pi$ should satisfy the properties of* perfect completeness*,* succinctness*, and* adaptive argument of knowledge for a oracle family $\mathbb{O}$ *specified as follows:*

- 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 $\lambda$), and*
  - $\mathsf{Verify}$ *runs in time* $\mathrm{poly}(\lambda + |\mathbb{x}|)$*.*
- Adaptive argument of knowledge for oracle family $\mathbb{O}$*: $\Pi$ satisfies* adaptive argument of knowledge *with respect to an oracle family $\mathbb{O}$ if for a randomly sampled CRS* $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$ *and oracle $\mathcal{O} \leftarrow \mathbb{O}$ and every efficient oracle adversary $\mathcal{A}^{\mathcal{O}}$ who makes $Q$ queries recorded in* query transcript $\mathsf{qt} = \{(q_i, \mathcal{O}q)\}_{i=1}^{Q}$*, there exists an efficient extractor $\mathcal{E}_\mathcal{A}$ given full access to the state of $\mathcal{A}$ including any random coins such that:*

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

The following theorem is a modification of [FN16, Theorem 6].

**Theorem 4.6.** *Let $\mathbb{O}$ be a random oracle family. If $\Pi$ is a SNARK, then $\Pi$ is an O-SNARK for $\mathbb{O}$.*

We justify our modification as follows. Though we omit auxillary inputs from our definition of an O-SNARK, we observe that the original theorem then implies that only every $\mathcal{U}_n$-auxillary input SNARK, where $\mathcal{U}_n$ is the uniform distribution over strings of length $n$, is a "no-auxillary input" O-SNARK for every random oracle family. We remark that $\mathcal{U}_n$ is considered a "benign" distribution, as discussed in [BP15; Bit+16], and that extraction is always possible in the presence of auxillary inputs sampled from benign distributions. Therefore, every SNARK is a $\mathcal{U}_n$-auxillary input SNARK and our modified theorem holds.

### 4.6 Consensus

Byzantine fault tolerance (BFT) refers to the ability of a computing system to endure arbitrary (i.e., Byzantine) failures of its components while taking actions critical to the system's operation. In this setting we are concerned with the challenge of state machine replication where a group of $n$ state machine replicas (validators) tries to solve the problem of deciding on a growing log of command requests (transactions) by clients. This can be achieved by running a Byzantine fault tolerant consensus mechanism. In this work we focus in particular on committee-based versions, such as as PBFT [CL99] and its derivatives [BKM18; KK+16; Mon20; Yin+19], which assume that $n > 3f$ where $f$ denotes the number of Byzantine replicas, i.e., those nodes under the control of an adversary. 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 $v$; otherwise, 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.

Every round of a PBFT-like algorithm has three phases: prepare, pre-commit, and commit. In the prepare phase, the current leader of the consensus committee proposes the next record that the system should agree upon. On receiving this prepare message, every replica validates the correctness of the proposal and broadcasts a pre-commit message to the group. The nodes wait until they collect a quorum of $(n - f)$ pre-commit messages and publish this observation with a commit message. Finally, replicas wait for a quorum of $(n - f)$ commit messages to make sure that enough of their peers have recorded the decision.

PBFT-like protocols rely upon a correct and live leader to start each round and only proceed if a two-thirds quorum exists. To ensure liveness in case a leader is faulty, PBFT has a view-change protocol. All nodes therefore monitor the leader's actions and initiate a view-change if they detect either Byzantine behavior or a lack of progress. Each node independently announces its desire to change leaders and stops validating the leader's actions. If a quorum of $(n - f)$ replicas decides that the current leader is faulty, then the next leader in a well-known schedule takes over.

## 5 Ultralight clients

In this section, we introduce a methodology for constructing ultralight clients for blockchains. We start by presenting a formalization of blockchain protocols as defined by their consensus algorithms. The main task of the ultralight client protocol designer is to define a "summary function" that, given a valid blockchain, extracts a constant-size "state" that alone is sufficient not only for checking whether new blocks are valid, but for updating itself. That is there should exist a "summary update function" that, given some state $s$ corresponding to a blockchain $b$, can determine if block(s) $u$ are a valid extension of $b$, and if so update $s$ to the $s'$ the summary function would output given $b||u$.

Often, just by having to satisfy the properties above, the state will contain enough information to permit the client access to blockchain-specific functionalities such as the ability make transactions, check account balances, etc.. Sometimes, additional information will need to be added to the state in order to facilitate certain blockchain interactions. We leave it to individual system designers define extensions to what we consider a minimal set of requirements for ultralight clients.

Ultimately, the ultralight client receives SNARKs proving knowledge of some new valid blocks (the witness) that result in a new state. The summary update function enables the prover to create incremental update proofs in time only quasi-linear in the new blocks. This is necessary because as a blockchain grows very large it becomes the case that CPU and memory requirements become insurmountable obstacles to proving the summary function, which runs in quasi-linear time in the entire blockchain. The use of SNARKs ensures that the ultralight client receives a succinct update proof it can verify in constant-time, and the knowledge-soundess property ensures the prover actually knows of valid update blocks that extend the previous blockchain resulting in the new claimed state. Starting with a trusted initial state (e.g., computed from the genesis block), the ultralight client can receive a series of incremenetal state update proofs and if they all accept we are guaranteed the existence of an extractor who can output a full blockchain corresponding to the final state of the ultralight client.

We proceed to present a formalization of this methodology. First, in Section 5.1 we describe the functions an ultralight client designer needs to define, and place some requirements on those functions. We also provide some discussion of techniques and tradeoffs for designing these functions we believe ultralight client designers will find helpful. Then, in Section 5.2 present a compiler that, given these functions and a SNARK, outputs a secure ultralight client that preserves the original consensus guarantees (i.e., an extractor can always extract a valid blockchain corresponding to any state an ultralight client is convinced to accept). Last, we define these functions for PLUMO and apply our compiler to obtain a secure ultralight client.

## 5.1 Functional components of an ultralight client

**Consensus Language.** The consensus algorithms of a blockchain define a *consensus language* $\mathcal{L}_C$ parameterized by some consensus parameters $\mathsf{cp}$, which might include descriptions of hash functions, difficulty, curve parameters, etc.. If $(\mathsf{cp}, b) \in \mathcal{L}_C$, then we say $b$ is a *valid blockchain* with respect to consensus parameters $\mathsf{cp}$ for the consensus language $\mathcal{L}_C$. For conciseness, we often write $b \in \mathcal{L}_C$ to mean $(\mathsf{cp}, b) \in \mathcal{L}_C$.

We require that a total order be defined on the set $\mathcal{L}_C$ (i.e., for every $b, b' \in \mathcal{L}_C$ either $b < b'$, $b > b'$, or $b = b$). Further, we require that the relation $\leq$ be efficiently computable. Most often, a total ordering of valid blockchains is defined by their length in blocks.

**Summary Function.** The purpose of an ultralight client is to reduce the storage, bandwidth, and computation requirements on a device, while still providing security guarantees with respect to an important subset of the interactions with a distributed ledger (e.g., the ability to make and receive transactions). The goal of an ultralight client system designer is to identify a constant-sized *state* that is sufficient, potentially together with additional prover help, to efficiently carry out these functionalities. We formalize this by means of an efficient *summary function* $S : \mathcal{L}_C \to \mathcal{L}_S$ that, given a valid blockchain $b$ outputs a state $s$ in the NP *state language* $\mathcal{L}_S$. This also gives the NP *state relation* $\mathcal{R}_S$ defined as

$$\mathcal{R}_S = \{(s;\ b) :\ s = S(b)\ \wedge\ b \in \mathcal{L}_C\}\ .$$

When given $b \notin \mathcal{L}_C$, the summary function $S$ must always output $\perp$.

We place to requirements on this state. First, given two states $s, s' \in \mathcal{L}_S$, an ultralight client must be able to efficiently decide which state is the canonical version, which no additional input. A total order on states is required that corresponds exactly to the total order on the corresponding blockchains (i.e., $s \leq s'$ implies $b \leq b'$). For most practical systems this is easily satisfied by having the summary function output the latest block number as part of the state.

**Incremental Proving.** A natural way to prove $s \in \mathcal{L}_S$ is to use a SNARK to prove knowledge of $b$ such that $(s, b) \in \mathcal{R}_S$. As noted, this becomes impractical as $b$ grows very large. To remediate this problem, we observe that for practical systems of interest[4] it is possible to define the summary function such that the state it produces is enough not only to check the validity of an *update* $u$ (i.e., a set of blocks that is a supposed extension of some chain), but to compute the state corresponding to $S(b\|u)$ if so. We formalize this by means of a requiring an efficient *update advice function* $U$ and an efficient *summary update function* $\hat{S} : \mathcal{L}_S \times \mathsf{Codomain}(U) \to \mathcal{L}_S$ such that :

- for all $(s, b) \in \mathcal{R}_S$ and updates $u \in \{0,1\}^*$ we have that $\hat{S}(s, \pi_u) = S(b\|u)$ given *update advice* $\pi_u \leftarrow U(s, u)$, and
- for every efficient adversary $\mathcal{A}$ there exists an efficient extractor $\mathcal{E}$ such that if $\mathcal{A}$, given uniformly sampled consensus parameters $\mathsf{cp}$, outputs $(b, \pi_u)$ such that $\hat{S}(S(b), \pi_u) = s' \neq \perp$, then $\mathcal{E}$, given full access to the state of $\mathcal{A}$ including any random coins, outputs $u$ such that $S(b\|u) = s'$.

So instead of receiving a SNARK proving knowledge of $b\|u$ such that $S(b\|u) = s'$, an ultralight starts with an initial trusted state $s_0 \in \mathcal{L}_S$ (with corresponding witness $b_0 \in \mathcal{L}_C$) and then receives a SNARK proving knowledge of $\pi_u$ such that $\hat{S}(s, \pi_u) = s_1$. That is, the SNARK is run to prove the relation

$$\mathcal{R}_{\hat{S}} = \Big\{ \big(s \in \mathcal{L}_S, s' \in \{0,1\}^*;\ \ \pi_u \in \mathsf{Codomain}(U)\big) :\ \ \hat{S}(s, \pi_u) = s' \Big\} \ .$$

The client may subsequently receive another SNARK updating the state from $s_1$ to $s_2$, and so on. Since knowledge-soundness composes, the knowledge-soundness requirements on the SNARK and on $\hat{S}$ guarantee that for accepting proofs we can extract $[u_i]_{i=1}^{n}$ such that $S(b_0\|u_1\| \ldots \|u_i) = s_i$ for $i \in [n]$. In this way we're able to guarantee the security and efficiency of the ultralight client state update protocol, while facilitating the incremental update proofs necessitated by large blockchains.

The reason we have defined $\hat{S}$ to operate on $\pi_u$ instead of $u$ directly, is that the $U$ function allows to us first apply specialized AoKs to the authenticated data in $u$. For example, in Plumo $U$ takes the multisignatures out of each block in $u$ and in their place appends a single aggregate multisignature (later we show how the security game for aggregate multisignatures can be re-cast as an AoK). Computing an aggregate multisignature is extremely cheap, but allows us shrink the $\hat{S}$ circuit nearly in half since verifying an aggregate multisignature on $n$ messages requires computing half the pairings verifying $n$ multisignatures does.

Generally speaking, there are many specialized AoKs with strictly linear time provers (with small constants), whereas SNARK provers run in quasi-linear time and require a costly reduction to R1CS. Ultralight client designers should seek to maximize use of highly prover-efficient AoKs in $U$, and subsequently verify those AoKs in $\hat{S}$.

As a last remark on the design of these functions, we observe some of the most efficient cryptography is pairing-based, including pretty much all the most efficient SNARKs. If a given blockchain uses pairing-based cryptography to authenticate data, then at minimum a two-chain

[4]Observe that systems in which, having previously verified $b \in \mathcal{L}_C$, it is not possible to verify $b\|u \in \mathcal{L}_C$ (and update the state) given some state $s = S(b)$ in time independent of $|b|$, are not "practical" in the sense that at a certain point the ledger grows long enough that most nodes lack the computational power to verify new blocks.

is required so that the pairing based authenticated data on the inner curve can be proven in the SNARK run on the outer curve. Two-chains have a very significant advantage over cycles in that they allow most cryptography in the system to be done over the normal size inner curve. For ultralight client designers that wish to produce incremental update proofs for much larger chunks of pairing-based authenticated data than we do in PLUMO, but want to preserve the advantages of using a two-chain, we recommend looking to a recent work that introduces a specialized—linear-time prover (with small constants), log-time verifier, and log-size proof—argument for all pairing-based languages [Bün+19c] whose inputs and proofs lie in the same curve.

**Preserving consensus soundness in the presence of oracles.** We want to show that the ultralight protocol as we have thus far described preserves the consensus guarantees (i.e., it is always possible to extract a valid blockchain for any accepting update proof). Consensus guarantees are generally formulated with respect to some cryptographic trapdoor functions. For example, without the corresponding secret key an efficient adversary cannot spend Bitcoin sent to a given public key. In our security definition, we give our adversary access to various oracles—signing, proving, etc.. It is indeed up to ultralight client system designer to enumerate them for their consensus algorithms.

We require that for any efficient adversary that can produce a SNARK updating an ultralight client's state, there exist an efficient extractor that, given given the query transcript $\mathsf{qt} = \{(q_i, \mathcal{O}(q_i))\}_{i=1}^{Q}$ of the adverary's oracle queries and responses and full access to its state including any random coins, outputs $u$ such that $S(b\|u) = s$. Unfortunately, we run into problems with the standard proof of knowledge definition for SNARKs when we introduce such oracles into the mix. 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].

Our compiler theorem requires that $(U, \hat{S})$ is an adaptive argument of knowledge for $\mathbb{O}$ and additionally $\Pi_{\mathsf{OS}}$ is an O-SNARK for $\mathbb{O}$. Since knowledge-soundness composes when the oracle provers are given access to oracles from the same oracle family, this ensures we are able to extract the witness $u$ even when it includes authenticated data. We reiterate is necessarily the task of the ultralight client designer both to define $\mathbb{O}$ according to their blockchain protocol and to prove that both $(U, \hat{S})$ and the SNARK they choose are adaptive arguments of knowledge for $\mathbb{O}$.

## 5.2 An ultralight client compiler

**Construction 1.** *Given a consensus language $\mathcal{L}_C$ with summary function $S$, update advice function $U$, summary update function $\hat{S}$, and SNARK $\Pi_{\mathsf{OS}} = (\mathsf{Setup}', \mathsf{Prove}', \mathsf{Verify}')$ we can construct a minimal ultralight client defined by a triple of efficient algorithms $\Pi_{\mathsf{UC}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ that work as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{crs}$*: given a security parameter $\lambda$ (in unary), outputs the common reference string* $\mathsf{crs} \leftarrow \mathsf{Setup}'(1^\lambda)$.
- $\mathsf{Prove}(\mathsf{crs}, s, u) \to (s', \pi)$*: the prover, given a common reference string* $\mathsf{crs}$*, state $s \in \mathcal{L}_S$, and update $u$, first runs $U(s, u)$ to obtain $\pi_u$. Next, they run $\hat{S}(s, \pi_u)$ to obtain $s'$. Finally, the prover runs* $\mathsf{Prove}'(\mathsf{crs}, (s, s'), \pi_u)$ *for the relation $\mathcal{R}_{\hat{S}}$ to obtain $\pi$. The prover outputs $(s', \pi)$.*
- $\mathsf{Verify}(\mathsf{crs}, s, s', \pi) \to \{0, 1\}$*: the verifier, given state $s \in \mathcal{L}_S$, claimed subsequent state $s'$, and proof $\pi$, outputs $1$ if $s' > s$ and* $\mathsf{Verify}'(\mathsf{crs}, s, s', \pi) = 1$*; else it outputs $0$.*

We next defines what it means for an ultralight client to be an adaptively secure.

**Definition 5.1** (Adaptively secure ultralight client)**.** *An ultralight client is* **adaptively secure** *with respect to an oracle family* $\mathbb{O}$ *if for a randomly sampled CRS* $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$ *and oracle* $\mathcal{O} \leftarrow \mathbb{O}$ *and every efficient oracle adversary* $\mathcal{A}^{\mathcal{O}}$ *who makes* $Q$ *queries recorded in* **query transcript** $\mathsf{qt} = \{(q_i, \mathcal{O}q)\}_{i=1}^{Q}$*, there exists an efficient extractor* $\mathcal{E}_{\mathcal{A}}$ *given full access to the state of* $\mathcal{A}$ *including any random coins such that:*

$$\Pr\left[\begin{array}{c|r} \mathsf{Verify}(\mathsf{crs}, S(b), s', \pi) = 1 & \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ \wedge & \mathcal{O} \leftarrow \mathbb{O} \\ S(b\|u) \neq s' & (b, s', \pi) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{crs}) \\ & u \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{crs}, \mathsf{qt}) \end{array}\right] \leq \mathrm{negl}(\lambda) \ .$$

We prove the following theorem demonstrating that Construction 1 defines an adapatively secure ultralight client under certain assumptions about it's functional components.

**Theorem 5.2.** *If* $(U, \hat{S})$ *is an adaptive argument of knowledge for* $\mathbb{O}$ *and* $\Pi_{\mathsf{OS}}$ *is an O-SNARK for* $\mathbb{O}$*, the ultralight client* $\Pi_{\mathsf{UC}}$ *output by Construction 1 is adaptively secure (Definition 5.1) with respect to* $\mathbb{O}$*.*

*Proof.* For every efficient adversary $\mathcal{A}$ that with non-negligible probability produces an output such that the $\mathsf{Verify}$ algorithm of $\Pi_{\mathsf{UC}}$ accepts in the context of the adaptive security game, we define an extractor $\mathcal{E}_{\mathcal{A}}$ as follows.

We begin by using $\mathcal{A}$ to build an adversary $\mathcal{B}$ against the O-SNARK $\Pi_{\mathsf{OS}}$ for oracle family $\mathbb{O}$ and relation $\mathcal{R}_S$. On input $\mathsf{crs}$ and given oracle access to $\mathcal{O}$, the adversary $\mathcal{B}$ runs $\mathcal{A}^{\mathcal{O}}(\mathsf{crs})$ to obtain $(b, s', \pi)$ (i.e., $\mathcal{B}$ passes through $\mathcal{A}$'s queries and responses to its own oracle) and then computes $s \leftarrow S(b)$. Adversary $\mathcal{B}$ then outputs $(s, s', \pi)$. By definition, any time the $\Pi_{\mathsf{UC}}$ verifier accepts given $(\mathsf{crs}, s, s', \pi)$, the $\Pi_{\mathsf{OS}}$ verifier does. So $\mathcal{B}$ makes the O-SNARK verifier accept with non-negligible probability. By the assumption that $\Pi_{\mathsf{OS}}$ is an O-SNARK for $\mathbb{O}$, there must exist an extractor $\mathcal{E}_{\mathcal{B}}$, that on input $\mathsf{crs}$ and $\mathsf{qt}$, outputs $\pi_u$ such that $\hat{S}(s, \pi_u) = s'$ with overwhelming probability whenever $\mathcal{B}$ gets the O-SNARK verifier to accept.

Next, we use $\mathcal{B}$ and $\mathcal{E}_{\mathcal{B}}$ to build an adversary $\mathcal{C}$ against $(U, \hat{S})$. Adversary $\mathcal{C}$, on input consensus parameters $\mathsf{cp}$ and oracle $\mathcal{O}$, samples $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$ and runs $\mathcal{B}^{\mathcal{O}}(\mathsf{crs})$ to obtain $(s, s', \pi)$. Adversary $\mathcal{C}$ also records both the blockchain $b$ that $\mathcal{A}$ outputs when run by $\mathcal{B}$ and also records the query transcript $\mathsf{qt}$. $\mathcal{C}$ then runs $\mathcal{E}_{\mathcal{B}}(\mathsf{crs}, \mathsf{qt})$ to obtain $\pi_u$, and finally outputs $(b, \pi_u)$. Since $\mathcal{B}$ gets the O-SNARK verifier to accept with non-negligible probability and when this happens $\mathcal{E}_{\mathcal{B}}$ outputs $\pi_u$ such that $\hat{S}(s(b), \pi_u) = s' \neq \perp$ (where we know $s'$ is not equal to $\perp$ because otherwise the $\Pi_{\mathsf{UC}}$ verifier would not have accepted), then $\mathcal{C}$ gets $\hat{S}$ to accept with non-negligible probability. By the assumption $(U, \hat{S})$ is an adaptive argument of knowledge for $\mathbb{O}$, there exists an extractor $\mathcal{E}_{\mathcal{C}}$ that outputs $u$ such that $S(b\|u) = s' \neq \perp$ with overwhelming probability whenever $\mathcal{C}$ gets $\hat{S}$ to output $s' \neq \perp$.

On input $\mathsf{crs}, \mathsf{qt}$ extractor $\mathcal{E}_{\mathcal{A}}$ runs extractor $\mathcal{E}_{\mathcal{C}}(\mathsf{qt})$ to obtain $u$, which it outputs. Since $\mathcal{E}_{\mathcal{C}}$ succeeds with overwhelming probability when $\mathcal{A}$ succeeds, we conclude that the ultralight client $\Pi_{\mathsf{UC}}$ is adpatively secure with respect to $\mathbb{O}$. $\square$

## 5.3 Plumo: an ultralight client

The following is a slight simplification of how these component functions are defined in Plumo. We describe the exact protocol in further depth in subsequent sections.

The summary function $S$ looks at the latest epoch message $m_n$ from a chain of valid epoch messages $(m_0\| \dots \|m_n) = b \in \mathcal{L}_C$ and returns the epoch message number corresponding to $m_n$ and the set of public keys signed in $m_n$.

The update advice function $U$ takes a set of public keys $s$ and a chain of epoch messages $u = (m_1, \dots, m\ell)$, strips out the multisignatures from each epoch message and then appends the aggregrate multisignature $\Sigma \leftarrow \mathsf{AggSign}(\mathsf{pp}, \{(\mathsf{apk}_i, m_i, \sigma_i)\}_{i=1}^{\ell})$. We prove the adaptive security of Plumo holds for any aggregrate multisignature scheme, so for now we don't worry about specifics.

The summary update function $\hat{S}$ takes a state $s$ consisting of an epoch message number $a$ and a set of public keys, and update advice $\pi_u$ consisting of one or more epoch messages and an aggregate multisignature over them. If $\pi_u$ contains a valid chain of epoch messages under the aggregate multisignature $\Sigma$, and the first epoch message in $u$ has epoch message number $a+1$ and an aggregate public key $\mathsf{apk}$ composed of the keys of $(2/3+1)$-majority of the public keys in $s$, then $\hat{S}$ outputs the public keys and epoch message number number of the last epoch message in $u$; else $\hat{S}$ outputs $\perp$.

The oracle family $\mathbb{O}$ reflects how the aggregate multisignature scheme used by the validators is the crucial trapdoor function used by $\mathbb{O}$. So sampling $\mathcal{O} \leftarrow \mathbb{O}$ is defined by running $\mathsf{KeyGen}(\mathsf{pp})$ to obtain a secret key $\mathsf{sk}$ stored by the $\mathcal{O}$, which on input a message $m$ outputs $\sigma \leftarrow \mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m)$.

We use the following informally stated result from [FN16], together with the observation that the aggregate multisignature security game can be re-cast as an adaptive proof of knowledge with respect to $\mathbb{O}$, in order claim that Plumo is adaptively secure for $\mathbb{O}$, given any SNARK $\Pi$ and any aggregrate multisignature scheme that's set in the hash-and-sign paradigm. The hash-and-sign paradigm is characterized by first hashing a message with a hash modeled as a random oracle, and then using the secret key to operate on that hash to produce the signature.

**Theorem 5.3** (Informal). *Let $\mathbb{O}$ be an oracle family defined by a computationally unforgeable signature scheme set in the hash-and-sign paradigm, where the hash function is modeled as a random oracle. Then all SNARKs are O-SNARKs for $\mathbb{O}$.*

## 6 Building blocks design

We now describe some of the design choices we've made in order to create a SNARK-friendly protocol. This is a set of techniques we've utilized in our concrete protocol instantiation described in 7.

**Choosing a multisignature scheme.** Multisignature schemes allow multiple parties to produce a single "multisignature" for a message $m$, attesting to the fact they each individually signed $m$. They were a natural design choice for our desiderata, allowing us to separate verification time and signature size from committee size.

While multisignatures for spans of epochs can be proved in a SNARK, when sending or checking for a transaction a client downloads the latest block header and verifies its multisignature directly. We use the BLS multisignature scheme [BLS01; BGS03; RY07] for both epoch messages and block headers, but instantiate the hash-to-curve function differently in the two settings, picking an optimized version for both outside and inside a SNARK.[5]

BLS multisignatures have small signature and public key size (a single group element each) and fast verification (two pairings). However, perhaps the most important property of BLS for Celo

[5]It is also possible to aggregate BLS multisignatures for multiple messages [BGS03]. Aggregate multisignatures are never consumed directly by any nodes on the network, but are used in our SNARK circuit to reduce prover time.

is that it supports *offline* aggregation of signatures into multisignatures. Indeed, virtually every efficient multisignature scheme requires multiple rounds of interaction [Dri+19], or in the case of the LOSSW multisignature scheme [Lu+13] interaction is not required, but each signer must know the final set of signers in advance.

For Celo's initial committee size of 100, with the right network incentives in place, it may happen that multisignatures are correctly computed on "first try" the vast majority of the time. However, Celo has a planned consensus algorithm upgrade in mind, BFTree,[6] that would greatly increase the number of validators. Not only does the increase in committee size exacerbate the failure potential here, but the BFTree protocol in its current form indeed relies on offline aggregation of signatures into multisignatures.

**Challenges of using BLS.** There are two primary complications that come with our use of BLS signatures. First, they are set in bilinear groups and so we must use a pairing-friendly curve. While pairing-friendly curves are less efficient at the same security level as their non-pairing-friendly counterparts, BLS signing, multisignature aggregation, and verification are still very fast. The real problem follows from the fact that the BLS verification equation cannot be efficiently proven in a SNARK over the same curve because of the mismatch between the field size and prime subgroup order.[7] While the same can be said for multisignatures schemes in the single group setting, the difference is that is very easy to find efficient *cycles of curves* used to resolve this problem when they need not all be pairing-friendly. Contrarily, known cycles of pairing-friendly curves are very inefficient, and indeed there is some evidence suggesting more efficient pairing-friendly cycles may not exist [CCW19].

Second, the security of BLS signatures holds in the random oracle model.[8] While certainly there are valid theoretical concerns about the random oracle model [CGH04], a more pertinent problem is that the "symmetric-flavor" cryptographic hash functions generally believed (i.e., for protocols of practical interest) capable of securely instantiating random oracles using Fiat-Shamir transformation [FS86] are very costly to prove inside SNARKs.

**A SNARK-friendly, composite algebraic-symmetric hash.** First, we address the need to instantiate the random oracle in BLS and the apparent lack of SNARK-efficient options.[9] Our solution to this problem is a composite algebraic-symmetric hash function that first uses BHPedersen, a SNARK-optimized variant [Hop+19] of the collision-resistant Pedersen hash [Ped92], to shrink the input to a single group element, and then calls Blake2Xs[10] [Aum+16] on that group element. By calling Blake2Xs on a much smaller input, less iterations of its internal compression function are required, which greatly increases prover efficiency. We show that if Blake2Xs is safe to instantiate a random oracle with, then our composite hash function is also safe to instantiate a random oracle with under the discrete-logarithm assumption.

**A two-chain of pairing-friendly curves.** As mentioned, BLS is positioned in the bilinear group

[6]See `https://medium.com/celohq/bftree-scaling-hotstuff-to-millions-of-validators-7d6930ee046a`.

[7]The inefficient way to do this would be to run a circuit generator such as [Ben+14b] on a representation of the verification program. Proving the resulting circuit would be much less efficient than using a pre-processing zkSNARK in second curve, which doesn't use complex data-dependent control flows or memory accesses like that output by the circuit generator would [Ben+14a].

[8]To our knowledge LOSSW is the only efficient multisignature scheme that is secure in the plain model [Dri+19].

[9]Recently introduced "algebraic-flavor" hash functions such as MiMC [Alb+16b] and Poseidon [Gra+19a] are very efficient to prove inside SNARKs and are suggested to be safe to instantiate a random oracle with. However, they have not yet received much cryptanalysis and we believe it ill-advised to deploy them until such time.

[10]We actually use a slight variant of Blake2Xs that saves us an additional compression function iteration.

setting. This means to efficiently prove the correctness of a batch of signatures we need a second pairing-friendly curve with a prime order subgroup equal to the size of the base field of the first. While there exist cycles of pairing-friendly curves where the base field of each curve is equal to the prime order subgroup of the previous forming a closed loop, and thus allowing polynomial-depth recursive composition of proofs, they are quite inefficient at reasonable security levels.

The most efficient known cycle at the 128-bit security level is a two-cycle of MNT curves each defined over a 753 bit field[11], currently deployed in the Coda protocol [Mec]. The large size of these curves is less important for the SNARK prover that runs intermittently on a powerful server, and more impactful for the validators who must constantly make signatures over this large curve. Especially in terms of supporting signing on CPU and memory-constrained devices such as hardware wallets, these large curves presented a number of practical hurdles to an implementation that could keep up with our target block production time of 5 seconds.

We resolved this problem by opting for a recently discovered "two-chain" of curves from [Bow+18]. While this pair of pairing-friendly curves only allows for a single level of recursion, the first curve is 377 bits, and has performance on-par with other pairing-friendly curves at the 128-bit level, allowing for reasonable performance on hardware wallets.

The second curve is 782 bits. We split epoch periods up into fixed-size chunks to prove—so a bootstrapping client might get a first SNARK proving 256 days, a second proving 32 days, and a third proving 8 days to get them up to speed 296 days after the network launch. While polynomial-depth recursion would keep this at a constant single SNARK, modern SNARKs are so small and fast to verify that having to verify a few was still acceptable even for our most limited clients.

**Splitting the relation.** In the BLS verification equation the only computational work besides checking a pairing equation is computing a hash-to-curve function (the random oracle) on the message. Following cues from Zexe, we first opted to compute the entire composite hash function on the messages in a SNARK over the inner $E_{\mathsf{BLS}}$ curve, and then prove it's correct verification over the larger, outer $E_{\mathsf{CP}}$ curve in addition to proving correctness of the pairing equation.

Surprisingly, this turned out to be computationally more expensive overall for our prover. We settled on implementing the Groth16 proof system [Gro16] because it's a better-studied SNARK with a verification equation requiring just 3 pairings, plus a linear number of exponentiations in the size of the instance. Surprisingly, It turns out that doing the hashing in $E_{\mathsf{CP}}$ instead of splitting the relation in this way is actually cheaper than the added cost of these exponentiations given the large size of epoch messages.

A known trick to reduce verifier time in Groth16 is to hash the instance down to a single field element, and modify the circuit to prove that the original computation is being performed on an opening of that hash of the correct form. The verifier then receives the original instance, hashes it themselves, and uses that hash to compute the single exponentiation needed to verify the SNARK. The problem with this method in our case is that the verifier is actually another SNARK prover, and again we run into the problem that hashing inside SNARKs is expensive. While only collision-resistance is needed here, using the more SNARK-friendly BHPedersen isn't feasible because it's not efficiently possible to use the same CRS of group elements in both curves.[12]

Ultimately, we found it most efficient BHPedersen over each message individually in $E_{\mathsf{CP}}$ and

[11]See `https://github.com/CodaProtocol/coda/blob/d57c4cc6935318541497dc6cbe0cedbfdd343c95/docs/specs/signatures/description.md`.

[12]We actually implement BHPedersen over "embedded" Edwards curves $E_{\mathsf{Ed/BLS}}$ and $E_{\mathsf{Ed/CP}}$ over $E_{\mathsf{BLS}}$ and $E_{\mathsf{CP}}$, respectively, where as noted in [Hop+19] a very prover-efficient circuit representation is permitted.

to run Blake2Xs over those messages in $E_{\mathsf{BLS}}$. The result of hashing each epoch message with BHPedersen over $E_{\mathsf{CP}}$ is interpreted as a string that we pack tightly into field elements over $E_{\mathsf{BLS}}$. This way we still do the more expensive part of the composite hash over the faster curve, but on a much smaller input, minimizing the cost of checking the proof over the inner curve in the outer curve.

**Final tweaks for a Plumo client.** Finally, we describe a few last techniques for minimizing client verification time and data costs. We first observe that the client only needs to know the start and end epoch messages the SNARK is proving there exists a valid chain between. Using the trick described above, we hash these two messages *individually* using Blake2Xs and pack the 512-bit total result into a two field element instance, minimizing the amount of exponentiation needed. The client, having already confirmed the chain up to epoch $a$, receives a 392 byte Groth16 proof over $E_{\mathsf{CP}}$ covering epoch $a$ to $b$, the 95 byte instance, and the $4,750$ byte epoch message[13] for epoch $b$. They hash epoch messages and confirm they match the instance and then verify the Groth16 proof.

As our SNARK circuits cover epoch chains of fixed powers two, it may require verifying multiple to get a client up to speed. Note that a client needs only check the initial epoch hash of the first SNARK instance and the final epoch hash of the final SNARK. For the intermediate hashes the client only makes sure that the line up such that the second instance element of the first SNARK is the same as the first instance element of the second SNARK and so on.

In all, a bootstrapping client needs only download the current epoch message (the genesis committee public keys being hardcoded) and a few SNARKs, which they batch verify.[14] The result is a PLUMO sync protocol which is extremely data and computation efficient.

## 7 Protocol Instantiation

Our construction is tailored for Celo's use-case which is a BFT-based chain. In the proof-of-stake protocol, epochs are about a day long (17280 blocks, each block targets 5 seconds), where at the last block of each epoch the validator set is updated. Our instantiation achieves light client synchronization efficiency, but also aims for efficiency for after the synchronization process. As long as they're continuously connected, they have to download and process subsequent blocks. This motivates being conservative with the size and performance of our signatures, while targeting high-security.

The epoch messages, which define the validator set changes, contain the full set of public keys to be used by validators. Being a long message, this motivates designing a SNARK-friendly hash-to-curve function. While SNARK-friendly hash functions such as MiMC [Alb+16a], Poseidon [Gra+19b] and Rescue [Aly+19] have been introduced in the last years, we chose to remain with more conservative choices based on established primitives, and using optimized versions of our chosen of primitives.

We discuss an instantiation of the BLS signature scheme that allows efficient verification inside SNARKs. To do so we need to have a mechanism to express relations involving pairing equations efficiently. We achieve this by using a SNARK-friendly hash-to-curve function and a pair of elliptic

[13]It is generally not necessary to even send the full epoch message. Validators may stay on consecutive committees as decided by the proof-of-stake mechanism such that bitmask and only the new public keys are sufficient to send in order for the client to reconstruct the epoch message.

[14]We use a few different well-known batching techniques including a variant of the small exponent test [BGR98; CL06] to reduce $3n$ pairings to $n + 2$, computing multi-exponentiations [Pip80], and computing a single final exponentiation after computing all Miller loops.

curves where BLS signatures are defined on the first and relations involving BLS signatures are expressed on the second. In addition, for SNARK proofs to be efficiently computable, we require that the base field of the BLS curve has high 2-adicity. For example, this makes the curve BLS12-381, that is currently being standardized in the IETF, unsuitable for our protocol [Bon+].

We make sure that the chosen solution allows the SNARK circuit to aggregate public keys by elliptic-curve point additions only, to achieve the best efficiency.

To avoid rogue key attacks, we sign a user-dependent string using a different domain in our hash function in order to create a proof-of-possession. As mentioned in [RY07], any string could work, as long as its known to the registration authority.

## 7.1 A SNARK-friendly Composite Hash-to-Curve Function

We use the try-and-increment method - we initialize a counter to 0 and prefix it to the message. If we fail at any of the steps described below, we increment the counter and try again.

**BHPedersenHash: CRH.** This collision-resistant hash had been introduced in [Hop+19] as a SNARK-friendly variant of a Pedersen hash. The hash uses a twisted Edwards curve having the same base field as the BLS curve. and benefits from the birational equivalence to a Montgomery curve to achieve greater efficiency

In this variant, the input message $m$ is divided into segments $s_i$, which are furthermore divided into 3-bit chunks $c_i$. The maximum number of chunks in a segment depends on the curve, and a formula to derive it is given in [Hop+19]. We denote it $C_{max}$.

We can formally describe its functionalities as follows:

**BHPedersenHash**

- $\mathsf{CRH.Setup}(1^\lambda, s) \rightarrow \mathsf{pp}_{\mathtt{CRH}}$
  1. Sample a group $(\mathbb{G}, q) \leftarrow \mathsf{SampleGroup}(1^\lambda)$.
  2. For $i \in \{1, ..., s\}$ Sample a segment generator $g_i \leftarrow G$.
  3. Output $\mathsf{pp}_{\mathtt{CRH}} := (\mathbb{G}, q, \{g_i\}_{i=1}^s)$.
- $\mathsf{CRH.Eval}(\mathsf{pp}_{\mathtt{CRH}}, m \in \{0,1\}^n) \rightarrow h$
  1. Parse $\mathsf{pp}_{\mathtt{CRH}}$ as $(\mathbb{G}, q, \{g_i\}_{i=1}^s)$.
  2. Divide the message into segments $m_i$ of size $C_{max}$.
  3. For each message segment $m_i$, divide the message into 3-bit chunks $m_{i,j}$. Furthermore, use the birational equivalence with a Montgomery curve on the generator $g_i$ to obtain $h_i$.
  4. For each chunk $m_{i,j} = (s_0, s_1, s_2)$, perform $(1 + s_0 + 2s_1)(1 - 2s_2) \cdot 2^{4i} h_i$ and sum them up into $f_i$.
  5. After processing each segment, use the birational equivalence on $f_i$ to obtain the equivalent Edwards point $e_i$. Sum all of these up into $e$.
  6. Output the x-coordinate of $e$.

BHPedersenHash is shown to be collision-resistant in [Hop+19]. It follows from the DLR assumption introduced in [Bün+18].

**Blake2Xs: XOF.** This cryptographic hash function is described in [Aum+16]. It uses at its base *Blake2s* through repeated hashing to obtain a digest of a desired size, making it an extensible-output function. In some instantiations, it's possible to optimize the hash even further by using a slight variant of Blake2Xs, where we do not perform the inner hash and instead use the output of BHPedersenHash. This is true when the base field of BHPedersenHash has size smaller than 512 bits. The XOF must output at least the amount of bits that can represent an element of the appropriate group, and an extra bit. This would be the modulus size of $\mathbb{G}_1$ if signatures are in $\mathbb{G}_1$ plus one. The extra bit is to determine the sign of $y$.

**Point derivation.** Following deriving enough random bits to attempt determining a point, we parse the bits (without the last one) as a possible $x$ coordinate. We then provide a matching $y$ coordinate such that $(x, y)$ is a valid curve point point. We use the last bit to efficiently check this is the correct y, by performing point compression and checking equality.

**Cofactor multiplication.** The last part is getting rid of low order components by multiplying by the cofactor. When signatures are in $\mathbb{G}_1$, we do it by directly multiplying by the cofactor. In $\mathbb{G}_2$, we've found it's more efficient to use the method from [BP17], where we multiply by a whole multiple of the cofactor.

## 7.2 SNARK-friendly BLS Signatures

One efficient instantiation that targets 128-bit security without compromising on efficiency is one that uses the curves from [Bow+18]. We describe it below.

**A pairing-friendly two-chain** We use a pairing-friendly two-chain introduced in [Bow+18], $E_{\mathsf{CP}}$ and $E_{\mathsf{BLS}}$. This two-chain a few properties that are advantegous to our protocol.

First, $E_{\mathsf{BLS}}$ (BLS12-377) is a curve from the BLS12 family, defined over a 377-bit prime, providing an approximate 128-bit security. This makes the curve appropriate for BLS signatures, even if the protocol wouldn't use SNARKs. $E_{\mathsf{CP}}$ is a curve built using the Cocks-Pinch method, having the base field prime of $E_{\mathsf{BLS}}$ as a factor in its group order. Second, since both $E_{\mathsf{CP}}$ and $E_{\mathsf{BLS}}$ have high 2-adicity, 46 and 47 respectively, allowing to efficiently prove large relations, as measured by amount of R1CS constraints. This is advantegous for our case, since we can use larger relations that span more epochs. Additionally, since proofs are generated only once in a while, different kinds of parties can generate these proofs. Parties with access to powerful hardware can generate these proofs faster.

**Public keys and signatures** Public keys are in $\mathbb{G}_1$ and signatures are in $\mathbb{G}_2$, chosen so as to minimize public key size.

To deal with rogue public key attacks, we note that in our case, the registration authority is an Ethereum smart contract. In the Celo blockchain, validators are associated with an Ethereum address, through which they register and to which they receive rewards. Since we wish to associate the BLS public key with the Ethereum address, we use the Ethereum address as the string in the proof-of-possession.

**Hash to curve** We use $E_{\mathsf{Ed/CP}}$ for BHPedersenHash, and use the variant of the XOF that doesn't perform inner hashing. This works since BLS12-377 has a base field of size 377 bits.

**Other instantiations** Our BLS signature construction works in any case it's possible to express relations involving pairing equations. Another example is [MS18], which uses a cycle of MNT curves of size about 753 bits, or a possible variant of Halo [BGH19] where one of the curves is pairing-friendly. In that case, the XOF variant that does use inner hashing is needed.

## 7.3 Plumo Client Sync Relation

We now present our relation for Plumo. We present two variants - one that is defined only in an $E_{\mathsf{CP}}$ SNARK and one that offloads some of the computation to a $E_{\mathsf{BLS}}$ SNARK. We use the Groth16 proof system for both SNARKs.

Let $\mathsf{AMS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{KeyAgg}, \mathsf{MultiSign}, \mathsf{Verify}, \mathsf{AggSign}, \mathsf{VerifyAgg})$ be an aggregate multisignature scheme with key space $\mathcal{K}$ and signature space $\mathcal{S}$. Let $v$ be the number of validators per-epoch and $\ell$ the number of epochs. Upon receiving an instance $\mathbb{x} = ([\mathsf{pk}_{i,0}]_{i=1}^{v},\ [\mathsf{pk}_{i,\ell}]_{i=1}^{v}, \mathsf{apk}_{\ell}, a)$ and a proof $\pi$ for the relation below, a client first confirms it know a chain of valid epoch messages up to epoch $a$, and that at epoch $a$ the current set of validator public keys are $[\mathsf{pk}_{i,0}]_{i=1}^{v}$. If this is true, the client checks that $\mathsf{KS\text{-}SNARK.Verify}() = 1$. For more details, the relation is shown in 7.4.

Offloading $\mathsf{Blake2Xs}$ computations to an inner proof is formally described by splitting the relation, such that the inputs to the inner proof are hashes of epochs obtained by running $\mathsf{BHPedersenHash}$ to obtain $h_i$ and the outputs from the inner proof are the outputs of $\mathsf{Blake2Xs}$ on each $h_i$. $\mathbb{G}_1$ and $\mathbb{G}_2$ below refer to the two source groups of $\mathsf{BLS12\text{-}377}$. Let $\eta \in \mathbb{Z}_m$ be a nonce. For more details, the split relation is shown in 7.4.

We note that although some of our primitives target 128-bit security, the final protocol will have less, due to Cheon's attack and its effect on big trusted setups, as noted in [Chi+19] and [GGW20], and NFS attacks, as noted in [BD19]. For a setup of around 180 epochs and our general use of $E_{\mathsf{BLS}}$ and $E_{\mathsf{CP}}$, we estimate the security would be between 114-bit and 120-bit.

## 7.4 Concrete relations

We first describe the non-split relation in 7.1.

Figure 7.1: Non-split relation

$$\mathcal{R}_{\mathsf{Abs}} \left\{ \begin{array}{ll} \multicolumn{2}{c}{\left( \begin{array}{c} \left([\mathsf{pk}_{i,0}]_{i=1}^{v},\ [\mathsf{pk}_{i,\ell}]_{i=1}^{v},\ \mathsf{apk}_{\ell}\right) \in \mathcal{K}^{v+v+1},\ \ a \in \mathbb{Z}_q\ ; \\ \left([[\mathsf{pk}_{i,j}]_{i=1}^{v}]_{j=1}^{\ell-1},\ [\mathsf{apk}_j]_{j=0}^{\ell-1}\right) \in \mathcal{K}^{v(\ell-1)+\ell},\ \ \Sigma \in \mathcal{S},\ \ [[b_{i,j}]_{i=1}^{v}]_{j=0}^{\ell-1} \in \{0,1\}^{v\ell}, \end{array} \right) :} \\ \mathsf{VerifyAgg}\left(\mathsf{pp}, [(\mathsf{apk}_j, m_j)]_{j=0}^{\ell-1}, \Sigma\right) & \text{where } m_j := \left(a+j, \mathsf{apk}_{j+1}, [\mathsf{pk}_{i,j+1}]_{i=1}^{v}\right) \\ \wedge\quad \mathsf{apk}_j = \mathsf{KeyAgg}(\mathsf{pp}, \{\mathsf{pk}_{i,j}\}_{i\in[v], b_{i,j}=1}) & \text{for } j \in \{0,1,\ldots,\ell-1\} \\ \wedge\quad \mathsf{apk}_{\ell} = \mathsf{KeyAgg}(\mathsf{pp}, [\mathsf{pk}_{i,\ell}]_{i=1}^{v}) & \\ \wedge\quad \mathsf{HammingWeight}(b_{1,j} b_{2,j} \cdots b_{v,j}) > \left\lceil \frac{2v}{3} \right\rceil & \text{for } j \in \{0,1,\ldots,\ell-1\} \end{array} \right\}$$

To describe the split relation, we start from an expanded description description of the non-split relation and move on to describe the split relation. We describe that in 7.2. over $E_{\mathsf{CP}}$.

Figure 7.2: Split relation

$$\left\{ \begin{array}{ll} \left( \begin{array}{c} \left([\mathsf{pk}_{i,0}]_{i=1}^{v},\ [\mathsf{pk}_{i,\ell}]_{i=1}^{v},\ \mathsf{apk}_\ell\right) \in \mathbb{G}_2^{v+v+1},\ a \in \mathbb{N}\ ; \\ \left([[\mathsf{pk}_{i,j}]_{i=1}^{v}]_{j=1}^{\ell-1},\ [\mathsf{apk}_j]_{j=0}^{\ell-1}\right) \in \mathbb{G}_2^{v(\ell-1)+\ell},\ \Sigma \in \mathbb{G}_1,\ [[b_{i,j}]_{i=1}^{v}]_{j=0}^{\ell-1} \in \{0,1\}^{v\ell},\ [\eta_j]_{j=0}^{\ell-1} \in \mathbb{Z}_m^{\ell} \end{array} \right) : & \\ e(\Sigma,\ G_2) = \sum_{j=0}^{\ell-1} e\left(\mathsf{H}_{\mathbb{G}_1}\left(\eta_j,\ a+j,\ \mathsf{apk}_{j+1},\ [\mathsf{pk}_{i,j+1}]_{i=1}^{v}\right),\ \mathsf{apk}_j\right) & \\ \wedge\quad \mathsf{apk}_j = \sum_{i=1}^{v} b_{i,j}\mathsf{pk}_{i,j} & \text{for } j \in \{0,1,\ldots,\ell-1\} \\ \wedge\quad \mathsf{apk}_\ell = \sum_{i=1}^{v} \mathsf{pk}_{i,\ell} & \\ \wedge\quad \mathsf{HammingWeight}(b_{1,j}b_{2,j}\cdots b_{v,j}) > \left\lceil \frac{2v}{3} \right\rceil & \text{for } j \in \{0,1,\ldots,\ell-1\} \end{array} \right\}$$

Splitting the relation, we can prove

$$\mathcal{R}_{\mathsf{H}_{\mathbb{G}_1}} := \left\{ \begin{array}{c} \left([m_j]_{j=0}^{\ell-1} \in \mathbb{G}_1^{\ell},\ [\eta_j]_{j=0}^{\ell-1} \in \mathbb{Z}_m^{\ell},\ a \in \mathbb{N},\ [(\mathsf{apk}_j,\ [\mathsf{pk}_{i,j}]_{i=1}^{v})]_{j=1}^{\ell} \in \mathbb{G}_2^{\ell+v\ell}\ ;\right) : \\ m_j = \mathsf{H}_{\mathbb{G}_1}\left(\eta_j,\ a+j,\ \mathsf{apk}_{j+1},\ [\mathsf{pk}_{i,j+1}]_{i=1}^{v}\right) \quad \text{for } j \in \{0,1,\ldots,\ell-1\} \end{array} \right\}$$

over BLS12-377 and

$$\left\{ \begin{array}{ll} \left( \begin{array}{c} \left([\mathsf{pk}_{i,0}]_{i=1}^{v},\ [\mathsf{pk}_{i,\ell}]_{i=1}^{v},\ \mathsf{apk}_\ell\right) \in \mathbb{G}_1^{v+v+1},\ a \in \mathbb{N}\ ; \\ \left([[\mathsf{pk}_{i,j}]_{i=1}^{v}]_{j=1}^{\ell-1},\ [\mathsf{apk}_j]_{j=0}^{\ell-1}\right) \in \mathbb{G}_2^{v(\ell-1)+\ell},\ \Sigma \in \mathbb{G}_1,\ [[b_{i,j}]_{i=1}^{v}]_{j=0}^{\ell-1} \in \{0,1\}^{v\ell}, \\ [\eta_j]_{j=0}^{\ell-1} \in \mathbb{Z}_m^{\ell},\ [m_j]_{j=0}^{\ell-1} \in \mathbb{G}_2^{\ell},\ \pi \in \mathbb{G}_1^2 \times \mathbb{G}_2 \end{array} \right) : & \\ \mathsf{Groth16.Vfy}(\mathcal{R}_{\mathsf{H}_{\mathbb{G}_1}},\ \mathbb{x},\ \pi) = 1 & \\ \quad \text{where } \mathbb{x} := \left([m_j]_{j=0}^{\ell-1},\ [\eta_j]_{j=0}^{\ell-1},\ a,\ [(\mathsf{apk}_j,\ [\mathsf{pk}_{i,j}]_{i=1}^{v})]_{j=0}^{\ell-1}\right) & \\ \wedge\quad e(\Sigma,\ G_2) = \sum_{j=0}^{\ell-1} e(m_j,\ \mathsf{apk}_j) & \\ \wedge\quad \mathsf{apk}_j = \sum_{i=1}^{v} b_{i,j}\mathsf{pk}_{i,j} & \text{for } j \in \{0,1,\ldots,\ell-1\} \\ \wedge\quad \mathsf{apk}_\ell = \sum_{i=1}^{v} \mathsf{pk}_{i,\ell} & \\ \wedge\quad \mathsf{HammingWeight}(b_{1,j}b_{2,j}\cdots b_{v,j}) > \left\lceil \frac{2v}{3} \right\rceil & \text{for } j \in \{0,1,\ldots,\ell-1\} \end{array} \right\}$$

## 8 Implementation

We've implemented our protocol using the Zexe set of libraries. While we do not use the DPC framework, we utilize both the native and circuit implementations of many cryptographic primitives, including the $E_{\mathsf{BLS}}$ implementation allowing for in-circuit SNARK verification, BLS signature verification and incomplete point additions and exponentiations. The code is available at [GS19]. Additionally, the code to perform setup for SNARKs on $E_{\mathsf{CP}}$ and $E_{\mathsf{BLS}}$ curves is available at [Kon20].

### 8.1 Techniques for circuit efficiency

As our circuit aims to compress a large history of the chain into a succinct proof, we paid special care for a circuit-efficient implementation of our cryptographic primitives. We describe below some of the techniques we've used.

**Epoch encoding** We utilize a circuit-friendly epoch encoding, which is a binary representation of the data needed for the light client protocol. This epoch encoding is signed by validators independently of the rest of the data during consensus, in order to both put the exact data that the circuit needs (i.e., removing parts that the circuit doesn't need and adding auxiliary data that makes the circuit work easier) and to use our efficient composite hash construction. Specifically, the data encoded for each epoch includes the epoch number, the compressed public keys of the new validator set, the aggregated public key of the new validator set and the threshold of required signatures.

**Point compression** Point compression in $\mathbb{G}_1$ and $\mathbb{G}_2$ is done by encoding the $y$ coordinate into a single bit. We take the common approach of using the bit to choose whether to take the bigger of the possible two $y$ options. In the circuit case, we check the equivalent condition of $y \leq \frac{p-1}{2}$ in $\mathbb{G}_1$ and $y_0 \leq \frac{p-1}{2} \vee (y_0 == \frac{p-1}{2} \wedge y_1 \leq \frac{p-1}{2})$ in $\mathbb{G}_2$, which is more efficient to implement, as it compares against constants.

**Hash to group** The epoch encoding is hashed into group in order to be signed. To hash into the group $\mathbb{G}_1$, where each element in defined by $(x, y) \in \mathbb{F}_p^2$, we first have to hash into the field $\mathbb{F}_p$. We compress the message using the collision-resistant $\mathsf{BHPedersenHash}$ and prepend a counter. On the short resulting string we use $\mathsf{Blake2Xs}$ with an XOF digest length that is large enough to be parsed as an $\mathbb{F}_p$ element, truncating bits to match the field element size and try parsing. If it fails, we try again. We continue by trying to parse the field element as a possible $x$ value. We attempt to find a matching $y$ and if we succeed, we multiply by the $\mathbb{G}_1$ cofactor to zero out low-order elements.

We've chosen to use this try-and-increment method rather than other existing constant time methods because of its efficiency inside a circuit. Other methods require hashing in any case, and add more operations afterwards. In our case, we provide the first counter that results in a hash that parses as a successful $x$ coordinate as a witness. The security of this relies on the fact that a threshold of honest validators would follow the try-and-increment process and only sign the message with the correct counter. Other methods, such as those described in [FH+], might be preferable to protect against implementation mistakes and DoS attacks.

**Bitmap and public key aggregation** The set of indices of the validators that participated in a signature is represented a set of bits of the same size as the validator set. We then utilize the bitmap in multiple ways: we sum the bits and compare that against the epoch threshold to check that enough validators signed, and we use it as the weights when aggregating the public keys to obtain the correct aggregated public key that would verify the signature.

We work with non-complete addition formulas, requiring that none of the addends are the neutral point or equal to each other. The validator set is of fixed size, and therefore it might have "holes" in it, where the entry is a neutral element. To ensure that aggregating the public keys does not result in the neutral element, we start the aggregation with the generator of the group as the first addend. This means that the generator is no longer considered a valid public key in our protocol.

**Edwards/Montgomery birational equivalence** The BHPedersen is defined over the $E_{\mathsf{Ed/CP}}$ curve, which is a twisted Edwards curve, and therefore it has complete addition formulas. The addition formulas cost 6 constraints per addition on the Edwards form. As done in [Hop+19], we use the birationally equivalent Montgomery form of the curve in a way that guarantees the incomplete addition formulas are enough, where these cost 3 constraints each. We additionally select appropriate parameters for the number of windows and chunks to guarantee collision resistance.

**Aggregated BLS signature** The epoch messages that we sign are distinct - at the very least, the epoch index makes them so. When verifying BLS signatures on distinct messages, it's possible to do it at the cost of $n+1$ pairings rather than the individual computation which would take $2n$ pairings, as mentioned in [BDN18]. This is done by computing an aggregate signature $\tilde{\sigma} = \sum_{i=1}^{n} \sigma_i$ and verifying (using the aggregated public keys $apk_i$): $e(\tilde{\sigma}, G_2) = e(H(m_1), apk_1) \cdots e(H(m_n), apk_n)$.

In our case, we provide the aggregated signature $\tilde{\sigma}$ as a witness, and as is commonly done we defer the final exponentiation until the end. Specifically, we calculate $\mathsf{FinalExponentiation}(\mathsf{MillerLoop}(\tilde{\sigma}, -G_2) \cdot \mathsf{MillerLoop}(H(m_1), apk_1) \cdots \mathsf{MillerLoop}(H(m_n), apk_n))$ and check whether it's equal to 1 in $\mathbb{G}_T$.

**First and last epoch output** The light client would eventually need to verify that the SNARK proof showed the evolution of the chain from the epoch that is input to the epoch that is output. That means that the light client needs access to this data. Directly outputting it as public SNARK inputs would result in multiple group exponentiations, which are costly. To cope with that, we hash using Blake2s both the input and output epochs, individually. We hash them individually to account for the case where multiple SNARK proofs are needed to synchronize the light client to the latest state. In this case, we do not need to know the pre-image of the intermediate epoch hashes, and just make sure that they line up.

## 8.2 Main circuit

We present a procedural description of our main circuit that implements our split relation in $E_{\mathsf{CP}}$. The second circuit, defined on $E_{\mathsf{BLS}}$, computes the second step of our composite hashes using Blake2Xs on each of the inputs and matches it with another corresponding input.

**Main circuit**

We first define the following helper method:

- $\mathsf{EncodeEpochToBits}(i, t, apk, \{pk\}_{i=1}^{n})$ :
  1. Encode $i$, the epoch index, as a 16-bit integer.
  2. Encode $t$, the required signer threshold, as a 32-bit integer.
  3. Encode $apk$, the aggregated public key of this validator set, as a compressed $\mathbb{G}_2$ point using the method from 8.1.
  4. Encode each public key in $\{pk\}_{i=1}^{n}$ as a $\mathbb{G}_2$ compressed point using the method from 8.1.

Then we describe the main circuit:

- $\mathsf{MainCircuit}(\sigma_{agg}, \pi_{hash}, \{bm_j\}_{j=1}^{N}, \{i_j, t_j, apk_j, \{pk_{j,k}\}_{k=1}^{n}\}_{j=1}^{N})$:
  1. Encode the input epoch into $e_0$ using EncodeEpochToBits, hash using Blake2s and save into $h_1$. Use the input epoch key set $\{pk\}_{i=1}^{n}$ as the next key set. Additionally
  2. Initialize $bh\_hashes := [], xof\_hashes := [], v := 1 \in \mathbb{G}_T$.
  3. For each epoch $e_i$, $j = 1...n$ perform:
     (a) Use the current key set $\{pk\}_{i=1}^{n}$ to build an aggregated public key $apk$ using the bitmap $bm_j$.

(b) Encode the epoch $e_j$ using $\mathsf{EncodeEpochToBits}$ and hash it using $\mathsf{BHPedersenHash}$, the first part of our composite hash. Accumulate into $bh\_hashes$. Additionally, compute the second part of our composite hashes which uses $\mathsf{Blake2Xs}$ and accumulate into $xof\_hashes$. Then complete the hash into group and denote it as $H(e_j)$.

(c) Compute $m := \mathsf{MillerLoop}(H(e_j), apk)$ and accumulate $v = v \cdot m$.

(d) Save $\{pk_{j,k}\}_{k=1}^{n}$ as the current key set.

4. Tightly pack $bh\_hashes$ and $xof\_hashes$ as elements of the scalar field of $E_{\mathsf{BLS}}$, and use $\mathsf{VerifyGroth16}$ with $\pi_{hash}$ to verify that each element in $bh\_hashes$ hashes into the corresponding element in $xof\_hashes$ when using $\mathsf{Blake2Xs}$.
5. Verify the aggregated BLS signature $\mathsf{FinalExponentiation}(\mathsf{MillerLoop}(\sigma_{agg}, -G_2) \cdot v) == 1$.

## 9 Evaluation

We benchmarked our split relation implementation on a Google Cloud machine with 4 Intel® Xeon® E7-8880 v4 processors and 3844 GB of DDR4 RAM. This kind of machine costs about \$6 an hour when run in a preemptible mode and \$25 an hour when run normally. For our use-case, this is attractive as proofs for a large span of epochs can be generated efficiently on such a machine, see Figures 9.1, 9.2 and 9.3. We report on the number of constraints for different epoch and consensus committee configurations in Table 1. Since the epoch length is about a day, proof generation machines can be turned off most of the time and be powered up only when there are sufficient epochs to generate a proof. Additionally, once a proof is created and gossiped through the network, everyone can verify it and start using it immediately, without the need for generating their own version of the proof. Summarizing the results, it is possible create proofs that span half a year worth of epochs for 100 validators in about 2 hours. We evaluated the performance of our verifier on a Motorola Moto G (2nd Gen), a 2014 mobile phone with 1GB RAM and a Quad-core 1.2 GHz Cortex-A7 processor. We used an unoptimized implementation, directly cross-compiled from [GS19]. The results show it is possible to verify such a proof in about 6 seconds.

In practice, we are going to deploy the non-split relation, since it simplifies the setup process to one setup for the $E_{\mathsf{CP}}$ curve SNARK, and the overhead is not big enough to justify the extra complexity.

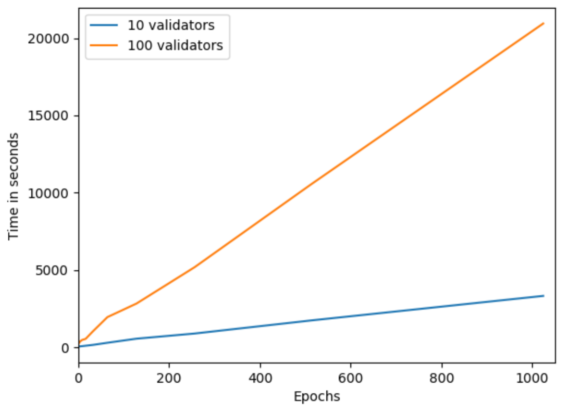

Figure 9.1: Proving time on $E_{\mathsf{CP}}$

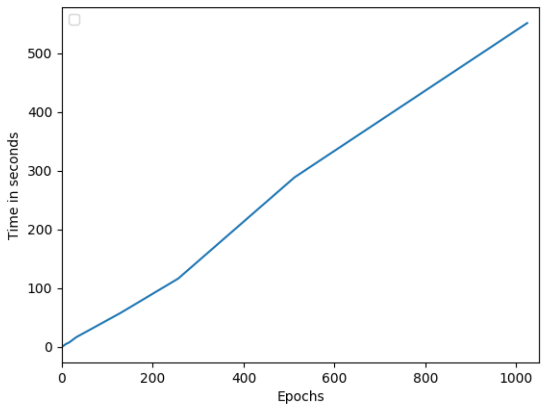

Figure 9.2: Proving time on $E_{\mathsf{BLS}}$

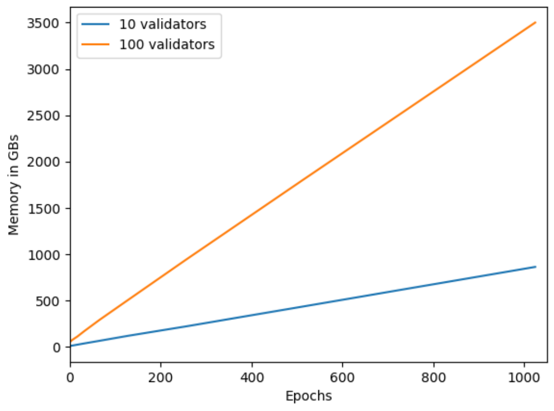

Figure 9.3: Peak memory consumption for proof generation

|  | 10 validators |  | 100 validators |  |
|---|---|---|---|---|
| Epochs | $E_{\mathsf{BLS}}$ | $E_{\mathsf{CP}}$ | $E_{\mathsf{BLS}}$ | $E_{\mathsf{CP}}$ |
| 1 | 44 722 | 882 845 | 44 722 | 7 258 711 |
| 2 | 89 190 | 944 285 | 89 190 | 7 684 723 |
| 4 | 178 126 | 1 067 165 | 178 126 | 8 536 747 |
| 8 | 355 998 | 1 312 925 | 355 998 | 10 240 795 |
| 16 | 711 742 | 1 804 445 | 711 742 | 13 648 891 |
| 32 | 1 423 230 | 2 787 485 | 1 423 230 | 20 465 083 |
| 64 | 2 846 460 | 4 753 568 | 2 846 460 | 34 097 470 |
| 128 | 5 692 920 | 8 685 734 | 5 692 920 | 61 362 244 |
| 256 | 11 385 586 | 16 550 063 | 11 385 586 | 115 891 789 |
| 512 | 22 770 918 | 32 278 721 | 22 770 918 | 224 950 879 |
| 1024 | 45 541 582 | 63 736 037 | 45 541 582 | 443 069 059 |

Table 1: Number of constraints for SNARK proofs

## 10 Future work

We've chosen to use the two-chain $E_{\mathsf{CP}}$ and $E_{\mathsf{BLS}}$ because of their unique suitability to our parameters - $E_{\mathsf{BLS}}$ is of appropriate size to achieve approximate 128−bit security, while $E_{\mathsf{CP}}$ allows proving statements that include pairings on $E_{\mathsf{BLS}}$. We use the Groth16 proof system. This approach has some downsides:

- $E_{\mathsf{CP}}$ is quite large and slow for the prover. The verifier also incurs some costs, which would be better with a smaller curve and a more efficient pairing.
- We must build the proof is one go over all the epochs - we can't incrementally add epochs to an existing proof.
- We must perform a trusted setup whenever the circuits change, which given our large circuits adds some operational overhead.

Future work could include using a recursive proof system of reasonable size, and ideally one without a trusted setup. This includes exploring how Fractal [COS19] and Halo [BGH19] could be used to iteratively prove statements as new epochs come into existence. Other possibilities that don't include recursion could be using a universal proof system, such as PLONK [GWC19] or Marlin [Chi+19], or even transparent ones such as [BFS19], [BS+18] or [KPV]. All of these come with trade-offs on prover complexity, proof size or verifier work.

An additional limitation is that our circuit is quite large, given the use of expensive hash functions and pairings. It might be possible to optimize this using hash functions that are designed to be circuit-friendly, such as MiMC [Alb+16a], Poseidon [Gra+19b] or Rescue [Aly+19]. These hash functions are gaining traction and their security is even being evaluated in hash challenges that pay bounties for finding collisions such as [Sta19] and [FL19]. An additional optimization could be done by using a recently published argument to reduce the cost of verifying the result of products of pairings [BMV].

## Acknowledgements

We thank Pratyush Mishra for helpful discussions.

## References

[Al-+18] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. "Chainspace: A Sharded Smart Contracts Platform". In: *Network and Distributed System Security Symposium, NDSS 2018*. 2018.

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

[Alb+16b] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. "Square Span Programs with Applications to Succinct NIZK Arguments". In: *Proceedings of the 20th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '16. 2016, pp. 191–219.

[Aly+19] A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec. *Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols*. Tech. rep. Cryptology ePrint Archive, Report 2019/426, 2019.

[Aum+16] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. *BLAKE2X*. `https://blake2.net/blake2x.pdf`. 2016.

[BD19] R. Barbulescu and S. Duquesne. "Updating key size estimations for pairings". In: *Journal of Cryptology* 32.4 (2019), pp. 1298–1336.

[BDN18] D. Boneh, M. Drijvers, and G. Neven. "Compact multi-signatures for smaller blockchains". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2018, pp. 435–464.

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

[Ben+14b] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture". In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security '14. 2014, pp. 781–796.

[Ben+19] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels. "Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1521–1538. ISBN: 9781450367479. DOI: `10.1145/3319535.3363221`.

[BFS19] B. Bünz, B. Fisch, and A. Szepieniec. *Transparent snarks from dark compilers*. Tech. rep. Cryptology ePrint Archive, Report 2019/1229, 2019.

[BGH19] S. Bowe, J. Grigg, and D. Hopwood. *Halo: Recursive Proof Composition without a Trusted Setup*. Tech. rep. Cryptology ePrint Archive, Report 2019/1021, 2019.

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

[BGS03] D. Boneh, C. Gentry, and H. Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *Proceedings of the 22nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '03. 2003, pp. 416–432.

[Bit+12] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. 2012, pp. 326–349.

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

[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.

[BKM18] E. Buchman, J. Kwon, and Z. Milosevic. *The latest gossip on BFT consensus.* 2018.

[BLS01] D. Boneh, B. Lynn, and H. Shacham. “Short Signatures from the Weil Pairing”. In: *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security.* ASIACRYPT ’01. 2001, pp. 514–532.

[BMV] B. Bünz, M. Maller, and N. Vesely. “Efficient Proofs for Pairing-Based Languages”. In: ().

[Bon+] D. Boneh, R. Wahby, S. Gorbunov, H. Wee, and Z. Zhang. *Work in Progress: draft-irtf-cfrg-bls-signature-00.*

[Bon+18] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. “Verifiable Delay Functions”. In: *Advances in Cryptology – CRYPTO 2018.* 2018.

[Bow+18] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. “Zexe: Enabling Decentralized Private Computation”. In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 962.

[BP15] E. Boyle and R. Pass. “Limits of Extractability Assumptions with Distributional Auxiliary Input”. In: *Proceedings of the 21st International Conference on the Theory and Application of Cryptology and Information Security.* ASIACRYPT ’15. 2015, pp. 236–261.

[BP17] A. Budroni and F. Pintore. “Efficient hash maps to $\mathbb{G}2$ on BLS curves”. In: *Cryptology ePrint Archive* 2017 (2017).

[BS+18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. “Scalable, transparent, and post-quantum secure computational integrity.” In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 46.

[Btc] *BTCRelay: A Bridge Between The Bitcoin Blockchain and Ethereum Smart Contracts.* 2014–2020.

[Bün+18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. “Bulletproofs: Short proofs for confidential transactions and more”. In: *2018 IEEE Symposium on Security and Privacy (SP).* IEEE. 2018, pp. 315–334.

[Bün+19a] B. Bünz, L. Kiffer, L. Luu, and M. Zamani. “FlyClient: Super-Light Clients for Cryptocurrencies”. In: (2019). https://eprint.iacr.org/2019/226.pdf.

[Bün+19b] B. Bünz, L. Kiffer, L. Luu, and M. Zamani. “Flyclient: Super-Light Clients for Cryptocurrencies.” In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 226.

[Bün+19c] B. Bünz, M. Maller, P. Mishra, and N. Vesely. *Proofs for Inner Pairing Products and Applications.* Cryptology ePrint Archive, Report 2019/1177. 2019.

[CCW18] A. Chiesa, L. Chua, and M. Weidner. “On cycles of pairing-friendly elliptic curves”. In: (2018). https://arxiv.org/abs/1803.02067.

[CCW19] A. Chiesa, L. Chua, and M. Weidner. “On Cycles of Pairing-Friendly Elliptic Curves”. In: 3.2 (2019), pp. 175–192.

[CGH04] R. Canetti, O. Goldreich, and S. Halevi. “The Random Oracle Methodology, Revisited”. In: *Journal of the ACM* 51.4 (2004), pp. 557–594.

[CGR11] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, Feb. 2011. ISBN: 978-3642152597.

[Chi+19] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. *Marlin: Preprocessing zksnarks with universal and updatable srs.* Tech. rep. Cryptology ePrint Archive, Report 2019/1047, 2019.

[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.

[CL99] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation.* 1999, pp. 173–186.

[Cor+13] J. C. Corbett et al. "Spanner: Google's Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013), 8:1–8:22. ISSN: 0734-2071. DOI: 10.1145/2491245. URL: http://doi.acm.org/10.1145/2491245.

[Cos] *Cosmos: The Internet of Blockchains.* 2017–2020.

[COS19] A Chiesa, D Ojha, and N Spooner. *Fractal: Post-quantum and transparent recursive proofs from holography.* Tech. rep. Cryptology ePrint Archive, Report 2019/1076, 2019.

[Dri+19] M. Drijvers et al. "On the Security of Two-Round Multi-Signatures". In: *Proceedings of the 40th IEEE Symposium on Security and Privacy.* S&P '19. 2019, pp. 1084–1101.

[EMSM19] C. Egger, P. Moreno-Sanchez, and M. Maffei. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 801–815. ISBN: 9781450367479. DOI: 10.1145/3319535.3345666.

[FH+] A. Faz-Hernandez, S. Scott, N. Sullivan, R. Wahby, and C. Wood. *Hashing to Elliptic Curves.*

[FL19] E. Foundation and P. Labs. *MiMC Hash Challenge.* http://mimchash.org/. 2019.

[FN16] D. Fiore and A. Nitulescu. "On the (In)Security of SNARKs in the Presence of Oracles". In: *Theory of Cryptography - 14th International Conference, TCC 2016-B, Part I.* Vol. 9985. Lecture Notes in Computer Science. 2016, pp. 108–138.

[FS86] A. Fiat and A. Shamir. "How to prove yourself: practical solutions to identification and signature problems". In: *Proceedings of the 6th Annual International Cryptology Conference.* CRYPTO '86. 1986, pp. 186–194.

[Gal+20] D. Galindo, J. Liu, M. Ordean, and J.-M. Wong. *Fully Distributed Verifiable Random Functions and their Application to Decentralised Random Beacons.* Cryptology ePrint Archive, Report 2020/096. 2020.

[Gen+13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: *Advances in Cryptology – EUROCRYPT 2013.* Springer Berlin Heidelberg, 2013, pp. 626–645.

[GGW20] K. Gurkan, A. Gabizon, and Z. Williamson. *Cheon's attack and its effect on the security of big trusted setups.* https://ethresear.ch/t/cheons-attack-and-its-effect-on-the-security-of-big-trusted-setups. 2020.

[Glu] A. Gluchowski. *Optimistic vs. ZK Rollup: Deep Dive.* URL: https://medium.com/matter-labs/optimistic-vs-zk-rollup-deep-dive-ea141e71e075.

[GPS08] S. D. Galbraith, K. G. Paterson, and N. P. Smart. "Pairings for Cryptographers". In: *Discrete Appl. Math.* 156.16 (2008), pp. 3113–3121. ISSN: 0166-218X. DOI: 10.1016/j.dam.2007.12.010.

[Gra+19a] 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: *IACR Cryptology ePrint Archive* 2019 (2019), p. 458.

[Gra+19b] 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: *IACR Cryptology ePrint Archive* 2019 (2019), p. 458.

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

[GS19] K. Gurkan and M. Straka. *bls-zexe*. `https://github.com/celo-org/bls-zexe`. 2019.

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

[GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Tech. rep. Cryptology ePrint Archive, Report 2019/953, 2019.

[Her18] M. Herlihy. "Atomic Cross-Chain Swaps". In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. PODC '18. Egham, United Kingdom: Association for Computing Machinery, 2018, pp. 245–254. ISBN: 9781450357951. DOI: `10.1145/3212734.3212736`.

[HLG19] E. Heilman, S. Lipmann, and S. Goldberg. *The Arwen Trading Protocols*. 2019.

[Hop+19] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. "Zcash Protocol Specification". In: (2019). `https://github.com/zcash/zips/blob/3c63d327a7208e659dac073485dda17a430bd1b9/protocol/protocol.pdf`.

[Int] *Interledger: An Open Protocol Suite for Sending Payments Across Different Ledgers*. 2016–2020.

[Kia+17] A. Kiayias, A. Russell, B. David, and R. Oliynykov. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *Advances in Cryptology – CRYPTO 2017*. 2017.

[KK+] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *IEEE Symposium on Security and Privacy (S&P) 2018*, pp. 19–34.

[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: *Proceedings of the 25th USENIX Conference on Security Symposium*. 2016.

[KLS16] A. Kiayias, N. Lamprou, and A.-P. Stouka. "Proofs of Proofs of Work with Sublinear Complexity". In: *Springer Berlin Heidelberg*. 2016, pp. 61–78.

[Kon20] G. Konstantopoulos. *snark-setup*. `https://github.com/celo-org/snark-setup`. 2020.

[KPV] A. Kattis, K. Panarin, and A. Vlasov. "REDSHIFT: Transparent SNARKs from List Polynomial Commitment IOPs". In: ().

[LNS20] J. Lee, K. Nikitin, and S. Setty. *Replicated state machines without replicated execution*. Cryptology ePrint Archive, Report 2020/195. 2020.

[Lu+13] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters. "Sequential Aggregate Signatures, Multisignatures, and Verifiably Encrypted Signatures Without Random Oracles". In: *Journal of Cryptology* 26.2 (2013), pp. 340–373.

[Mal+17] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. "Concurrency and Privacy with Payment-Channel Networks". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 455–471. ISBN: 9781450349468. DOI: `10.1145/3133956.3134096`.

[Mal+20] G. Malavolta, P. M. Sanchez, C. Schneidewind, A. Kate, and M. Maffei. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". In: *Proceedings of the 2020 Network and Distributed Systems Security Symposium*. NDSS '20. 2020.

[Mec] *Coda: Decentralized cryptocurrency at scale.* `https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf`. 2018.

[Mon20] H. Moniz. *The Istanbul BFT Consensus Algorithm.* 2020. arXiv: `2002.03613 [cs.DC]`.

[MS18] I. Meckler and E. Shapiro. "Coda: Decentralized cryptocurrency at scale". In: (2018).

[Nak09] S. Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* Tech. rep. Manubot, 2009.

[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 17).* USENIX Association, 2017, pp. 1271–1287. ISBN: 978-1-931971-40-9.

[PB17] J. Poon and V. Buterin. "Plasma: Scalable Autonomous Smart Contracts". In: (2017). https://www.plasma.io/plas

[Ped92] T. P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *Proceedings of the 11th Annual International Cryptology Conference.* CRYPTO '91. 1992, pp. 129–140.

[Pip80] N. Pippenger. "On the Evaluation of Powers and Monomials". In: *SIAM Journal on Computing* 9.2 (1980). Preliminary version appeared in STOC '76., pp. 230–250.

[Pol] *Polkadot: United Networks of State Machines.* 2018–2020.

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

[Sta19] StarkWare. *StarkWare Hash Challenge.* `https://starkware.co/hash-challenge/`. 2019.

[Syt+17] E. Syta et al. "Scalable Bias-Resistant Distributed Randomness". In: *38th IEEE Symposium on Security and Privacy.* San Jose, CA, May 2017.

[TMSM19] E. Tairi, P. Moreno-Sanchez, and M. Maffei. *$A^2L$: Anonymous Atomic Locks for Scalability and Interoperability in Payment Channel Hubs.* Cryptology ePrint Archive, Report 2019/589. 2019.

[TSB19] J. Teutsch, M. Straka, and D. Boneh. *Retrofitting a two-way peg between blockchains.* 2019. arXiv: `1908.03999 [cs.CR]`.

[Val08] P. Valiant. "Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency". In: *Theory of Cryptography.* Springer Berlin Heidelberg, 2008, pp. 1–18.

[Wze] *Wrapped ZEC.* 2019-2020.

[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 (PODC) 2019.* 2019, pp. 347–356.

[Zam+19] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt. "XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets". In: *2019 IEEE Symposium on Security and Privacy.* SP '19'. 2019, pp. 193–210. DOI: `10.1109/SP.2019.00085`.

[Zam+19] A. Zamyatin et al. *SoK: Communication Across Distributed Ledgers.* Cryptology ePrint Archive, Report 2019/1128. 2019.