Trustless Efficient Light Clients

Yuxiang Qiu

MEng Computer Science

Supervisors

Prof. Philipp Jovanovic and Dr. Alberto Sonnino

April 2025

Abstract

Light clients play a crucial role in blockchain ecosystems by enabling devices with limited computational resources to participate in on-chain activities. However, achieving full security typically requires downloading data that scales linearly with the blockchain's size. This poses a major scalability challenge as blockchains grow, ultimately undermining the very utility of light clients.

In this work, we propose **Mim**, a system that leverages *succinct non-interactive arguments of knowledge* (SNARKs) to efficiently prove the correctness of committee rotations in Byzantine Fault Tolerant (BFT)-based light client protocol. Unlike prior approaches that either compress downloadable data or redesign consensus mechanisms around zero-knowledge proofs, our solution is generalizable and achieves verification performance that is independent of or sub-linear in the chain size.

Our core contribution is the use of *folding-based SNARKs*, a recent advancement that enables multiple instances of identical NP statements - in our case, committee rotation verifications - to be folded into a single proof. When paired with an appropriate proving system, this allows us to generate constant-size $O(1)$ proofs of committee rotation. To further enhance efficiency, we introduce a novel data structure called the **Levelled Merkle Forest (LMF)**, which mitigates the need to run SNARKs for every committee rotation. LMF accumulates committee data into hierarchical Merkle trees, enabling variable-length proofs post-verification.

We evaluate our design on a simulated blockchain modelled after the checkpoint structure of Sui. Our extrapolated results show that, given sufficient memory, it is feasible to generate a committee rotation proof in under half a day (without LMF), faster than the typical one-day checkpoint generation interval. Moreover, with LMF, proof covering one year of checkpoint data can be produced in under a week. Concurrently, LMF achieves performance comparable to the Merkle tree while reducing proof sizes for all of the committed values, making it a practical and efficient alternative. We conclude by highlighting several promising directions for future work to further enhance the efficiency and applicability of our system.

Contents

Acknowledgments

I would like to express my deepest gratitude to my supervisors, Prof. Philipp Jovanovic and Dr. Alberto Sonnino, for their invaluable guidance, unwavering support, and insightful feedback throughout the research and writing of this thesis. Their expertise and constructive criticism have played a crucial role in shaping this work.

I also extend my sincere thanks to Alberto and Mysten Labs for their generous support in providing AWS resources for the evaluations conducted in this study.

Finally, I am profoundly grateful to my partner and family for their unconditional love, encouragement, and sacrifices. Their belief in me has been a constant source of motivation and has enabled me to reach this significant milestone.

This thesis would not have been possible without the contributions and support of all these individuals. Thank you!

Chapter 1

Introduction

1.1 Problem Outline

Since the introduction of Bitcoin, blockchain technology has evolved into a transformative force across various industries and communities. Initially popularized as the underlying framework for cryptocurrencies [YW18], blockchain now powers a wide range of applications, such as decentralized finance (DeFi) [WPG+23] and healthcare [AME19]. At its core, blockchain is a distributed system in which each node maintains a copy of all transactions. As new transactions are added, nodes reach consensus according to a defined protocol, extending the blockchain.

As blockchain technology continues to mature, various types of clients have emerged, generally falling into the following categories. *Full nodes* ensure the security of the blockchain by executing a consensus protocol to agree on the blockchain's state. They store a complete copy of the blockchain, including all historical data, and communicate with other full nodes using the gossip protocol. These nodes possess the necessary resources to handle the storage, bandwidth, and computational demands associated with maintaining the full blockchain history. In contrast, *light clients* are resource-constrained devices, such as mobile phones or web browsers, which do not store the entire blockchain and lack the computational power to perform heavy tasks. Instead, they rely on full nodes to serve as intermediaries, fetching blockchain data and submitting transactions on their behalf [CBC22].

For instance, the Simplified Payment Verification (SPV) protocol, as proposed in the Bitcoin whitepaper [Nak08], enables light clients to verify transactions without downloading the full blockchain. Similarly, Ethereum [ACP+24] employs a light-client protocol that uses checkpoints and a committee-based mechanism to address

the blockchain's fast block generation rate.

Despite being termed light client protocols, these protocols can still be too resource-intensive for certain lightweight environments. Light clients often need to download some data, such as the chain of block headers in SPV, which grows linearly with the size of the blockchain. After downloading this data, clients must also verify that the chain of headers begins at the genesis block and that each block has been correctly validated by consensus nodes. This verification process is similarly linear in the size of the chain. Given that many modern blockchains exhibit high block generation rates [NAK+22] and that light clients frequently operate in offline phases [CBC22], this approach becomes impractical in resource-constrained settings.

1.2 Our Approach

This work addresses the aforementioned challenges by developing solutions that enable light clients to efficiently download and verify data, with verification time independent of the blockchain's size. The primary objective is to design a secure protocol that facilitates succinct verification of blockchain states for light clients. Specifically, we focus on a Byzantine Fault Tolerant (BFT)-based light client protocol in which a regularly rotated committee maintains a chain of checkpoints. Each checkpoint includes signatures from a supermajority of the current committee, along with transaction data and information about the subsequent committee. In this protocol, the light client must track committee rotations to verify the corresponding checkpoints. This design, adopted by several prominent blockchains [ACP+24, RML], plays a critical role in ensuring both the security and efficiency of light client protocols.

At the core of our approach lies the cryptographic primitive known as *Succinct Non-Interactive Arguments of Knowledge* (SNARK). SNARK provides a complete, knowledge-sound, and succinct proof system for relations expressed as arithmetic circuits. Notably, they can produce proofs as short as three group elements, regardless of the circuit's complexity [Gro16].

To prove committee rotations, we employ a folding-based SNARK - a variant of recursive SNARK - that enables the prover to fold multiple instances of a relation into a single instance. The prover then demonstrates the correctness of the folded instance, which implicitly proves the validity of all individual (unfolded) instances. This mechanism allows for the aggregation of multiple committee rotation verifications into a single proof, significantly improving proving efficiency.

To further accelerate the proving process, we introduce a novel cryptographic accu-

mulator, the **Levelled Merkle Forest**, which is integrated into the circuit to verify the correct construction of the accumulator. This structure enables efficient proof generation for the existence of an intermediate committee, given a trusted starting committee (assumed by all light clients) and an ending committee (which the light client can verify using proof generated by the folding-based SNARK).

We implement our protocol, including the newly introduced LMF data structure, in Rust with over 8000 lines of code. During this process, we implemented the first open-source, generic R1CS implementation of the hash-to-curve algorithm in `arkworks`, and fixed a critical bug in the emulated field variable that generated unsatisfiable constraints.

In our evaluation, we show, by extrapolation, that without LMF, our light client protocol can prove committee rotation in under half a day given sufficient memory - faster than the typical one-day checkpoint generation time - while maintaining a constant verification time of around 3 seconds for a committee of 512 members, regardless of checkpoint size. With LMF, the protocol can produce proof covering one year of checkpoint data in under a week. We also estimate the memory requirements needed to deploy the protocol in practice. Simultaneously, we demonstrate that LMF offers performance comparable to traditional Merkle trees, while its variable-length proof scheme reduces proof sizes for all of the committed values.

1.3 Contributions

In summary, our contributions are as follows:

- We present, to the best of our knowledge, the first open-source, generic hash-to-curve implementation for BLS12 curves within the `arkworks` ecosystem. Our implementation supports the [WB19] hash-to-curve algorithm and can be used inside any R1CS gadget defined in `arkworks`.
- We identify and resolve a bug in the widely-used SNARK library `arkworks`. This bug affected the generation of satisfiable constraints during synthesis for emulated field variables. Our fix enables correct and reliable use of emulated field variables in `arkworks`.
- We design a protocol that enables light clients to efficiently verify the security of received data using folding-based SNARK. Our construction builds on the security of the underlying IVC scheme, SNARK scheme, and the light client protocol.

- We introduce a novel data structure - **Levelled Merkle Forest (LMF)** - to accelerate proof generation for folding-based SNARK. This structure, which may be of independent interest, offers a new paradigm for accumulator design: it uses a small amount of extra space (constant outside the R1CS circuit and logarithmic inside) to enable variable-length proofs. We formally and experimentally analyze the time and space complexity of both the on-circuit and off-circuit operations of this data structure.

Some of the techniques discussed in this paper have broader applications beyond committee rotation verification. For example, the hash-to-curve implementation can be applied whenever there is a need to hash into BLS12 group elements. Similarly, the LMF data structure can be used as a standard method for constructing Merkle trees in R1CS, especially when variable-length proofs are favoured. However, this paper primarily focuses on the verification of committee rotations.

1.4 Overview of Chapters

Chapter 2 provides an overview of the notation used throughout this work, along with a summary of the background and relevant research.

Chapter 3 reviews key works in the field, analyzing how prior research addresses the challenges identified in this study.

Chapter 4 introduces the blockchain abstraction used in this work, followed by the design and security proof of the light client protocol. It describes the interface of the folding-based SNARK circuit and outlines the design of BLS signature and committee rotation verification circuits. It also presents the new data structure, Levelled Merkle Forest (LMF), detailing its algorithms and analyzing the time and space complexity of both on- and off-circuit operations. The chapter concludes with the final light client protocol design incorporating LMF.

Chapter 5 covers implementation challenges, with a focus on the R1CS implementation of hash-to-curve. It discusses how a bug in the `arkworks` library was diagnosed and resolved to avoid unsatisfiable constraints, and highlights circuit-level optimizations that reduce constraint count.

Chapter 6 evaluates the system, analyzing the time and memory costs of the light client protocols (with and without LMF) and comparing LMF performance against traditional Merkle trees.

Chapter 7 examines the security of the light client protocol in the presence of forks

and discusses its practical deployment. It also outlines potential future research directions to improve the system's applicability, highlighting the associated challenges.

Chapter 8 concludes the paper, summarizing the key findings and contributions.

Chapter 2

Background

In this chapter, we cover the notations and essential background on blockchains, elliptic curves, BLS signatures, accumulator schemes, IVC, and SNARK.

2.1 Notation

We use F_q to denote a finite field of size q. Let $r \xleftarrow{\$} F_q$ denote drawing a random value from the finite field F_q. We use bold lowercase letters for vectors and bold uppercase letters for matrices. For a vector $\mathbf{v} \in F^m$, we denote the elements of $\mathbf{v}$ as $(v_0, \ldots, v_{m-1})$ and use $\mathbf{v}_i$ to index the ith element of $\mathbf{v}$.

2.2 Blockchain

A blockchain is a distributed system where a group of nodes maintains a copy of an ordered list of blocks. A block is a data structure consisting of a header and a list of transactions. The block header contains metadata about the block, including the hash of the previous block and validity proofs, such as solutions to hash puzzles in proof-of-work systems.

2.2.1 BFT-Based Light Client Protocol

In a blockchain network, different types of clients exist, such as full nodes and light clients. Full nodes store the entire history of the blockchain and are responsible for verifying transaction execution and blockchain consensus. In contrast, light clients rely on full nodes to fetch the desired blocks and can only verify consensus using the additional information provided by the full nodes.

The type of information supplied by full nodes depends on the underlying consensus protocol. For instance, in the original Bitcoin SPV protocol [Nak08], full nodes provide all block headers, starting from the genesis block and up to the block requested by the user. To verify the header, the light client begins at the genesis block, checks whether the next block includes a hash to the previous block, and verifies the validity proof in the block header.

In this paper, we focus on a different type of light client protocol: the BFT-based light client protocol. In a BFT-based protocol, a committee of validators regularly maintains a chain of checkpoints. Each checkpoint, denoted as `cp`, includes information such as transaction data (typically represented as a Merkle tree root) and public keys (with weights) of the next committee. At each round r, the committee collectively determines the next checkpoint, denoted as cp_r. For a proposed checkpoint to be valid, it must receive signatures from a supermajority (greater than $\frac{2}{3}$) of the committee members. The committee is rotated at regular intervals as defined by the protocol to ensure both security and liveness, which is why the checkpoint includes information about the next committee.

In such a light client protocol, the light client must verify the validator signatures on the proposed checkpoints and track committee rotations. To manage this, the light client typically maintains a state, denoted as S_r, at round r, which tracks the committee for that round, including the public keys (with weights) of the committee members. This state is updated using the public key information from the newly verified checkpoints.

Definition 1 (Security of BFT-Based Light Client Protocol)**.** *For this paper, we are concerned with two properties of the BFT-based light client protocol:*

- ***Security:***
  - *Committee:*
    - *The initial committee is set up in a trusted manner.*
    - *If, at round r, a supermajority of the committee signs a checkpoint cp_{r+1}, cp_{r+1} will be the next checkpoint. This means that 1) cp_{r+1} points to the most recent valid checkpoint, contains valid transaction data, and correct public keys (with weights) [BDN18] about the next committee and 2) cp_{r+1} will be the only checkpoint signed by the supermajority of the committee at round r.*
  - *Client: If, with state S_{r-1}, a light client accepts cp_r as a valid next check-*

point, this means that cp_r is a checkpoint signed by the supermajority of the committee at round $r-1$.

- ***Liveness:*** *If an honest full node receives some transaction at some round, then the supermajority of the committee will approve a checkpoint containing that transaction [XZC+22].*

2.3 Elliptic Curves

An elliptic curve group $E(F_q)$ is a group specified by a set of points $P = (x, y)$ over a base field F_q and a scalar field F_r, satisfying elliptic curve equations. It contains a neutral element O (point at infinity) and supports point addition and scalar multiplication.

2.3.1 Pairing

Some elliptic curve additionally supports a pairing structure, defined as follows.

Definition 2 (Bilinear Pairing)**.** *Consider a tuple $(p, G_1, G_2, G_T, e, g, h)$ where p is a prime number, G_1, G_2, G_T are groups of prime order p, g is the generator for G_1 and h is the generator for G_2. A pairing is a map $e : G_1 \times G_2 \to G_T$ that satisfies the following properties*

- ***Bilinearity****: $\forall a, b \in F_r, P \in G_1, Q \in G_2, e(aP, bQ) = e(P, Q)^{ab} = e(abP, Q) = e(P, abQ)$*
- ***Computability****: There are efficient algorithms for computing e.*

In our work, we consider a particular family of elliptic curves that support pairing - BLS12 curves.

2.3.2 Cofactor

The cofactor of a subgroup is defined as the ratio between the order of the full elliptic curve group and that of the subgroup. In traditional elliptic curve cryptography, it is crucial for the cofactor to be small - ideally one - to mitigate small subgroup attacks on the discrete logarithm problem. However, for elliptic curve groups that support pairings, such as those used in pairing-based cryptography, the cofactor can be substantially larger [Edg15].

To mitigate small subgroup attacks in pairing-friendly elliptic curve groups, cofactor clearing is required. The straightforward approach involves multiplying a given curve

point by the cofactor h, thereby eliminating any h-torsion components [Bow19]. However, because the cofactors in these groups are typically large, this naive method is computationally inefficient. Recent advances have demonstrated that, for certain families of pairing-friendly curves, cofactor clearing can be performed much more efficiently using endomorphisms. This optimized approach has been standardized and is now the recommended method for cofactor clearing [FHSS+23].

2.3.3 Hash to Curve

Definition 3 (Hash to Curve)**.** *A hash-to-curve function is a deterministic function that maps an arbitrary input to a point on $E(F_q)$. Formally, given a cryptographic hash function $h : \{0,1\}^* \to \{0,1\}^k$ and an elliptic curve E defined over a finite field F_q, a hash-to-curve function $H : F_q^* \to E(F_q)$ takes an input string m, processes it through h, and outputs a point $P \in E(F_q)$ such that the mapping is one-way, collision-resistant, and indistinguishable from a uniform random point on E.*

The hash-to-curve algorithm proposed in [WB19] is the standard method widely used for hashing to BLS12 elliptic curves.

2.4 BLS Signature

BLS signature [BLS01] is a digital signature based on bilinear pairing.

Definition 4 (BLS Signature)**.** *Consider a tuple $(p, G_1, G_2, G_T, e, g, h)$ that supports bilinear pairing. BLS signature is a tuple of efficient algorithms (Keygen, Sign, Verify) that are specified as follows:*

- *Keygen(λ): Pick $sk \xleftarrow{\$} F_r$ as the private key and the public key is $pk = sk \cdot g \in G_1$.*
- *Sign(sk, m): $sk \cdot H(m)$, where H is a cryptographic hash that hashes the message to G_2.*
- *Verify(σ, m) $\to \{0,1\}$: Return 1 if $e(pk, H(m)) = e(g, \sigma)$ else 0.*

Note that the above definition assigns public keys to G_1 and signatures to G_2, but they can also be assigned in the reverse order.

In addition, BLS Signature can be aggregated. Given public keys $\{sk_i \cdot g\}_{i=1,\ldots,t}$, one can compute an aggregate public key $pk = \sum_{i=1}^{t} sk_i \cdot g$. To verify a t-multisignature σ on a message m given aggregate public key pk, the verifier checks whether $e(g, \sigma) =$

$e(pk, H(m))$ [PSG+24]. If pk is provably the sum of t individual BLS public keys, we can be assured that t parties signed the message [BDN18].

In this paper, we focus on the scenario where the light client protocol uses the BLS signature on top of the BLS12 curves. However many techniques developed in this paper can be applied generally to any other signatures.

2.5 Accumulator Scheme

A cryptographic accumulator [BdM94] is a mechanism that commits to a collection of values by producing a succinct digest. This digest is binding, meaning it is computationally infeasible to construct two distinct collections that yield the same digest. Additionally, many accumulators support succinct membership proofs, allowing efficient verification that a specific value is included in the committed set.

Definition 5 (Merkle Tree)**.** *A Merkle tree [Mer88] is a vector accumulator. Given a vector* $\mathbf{v}$ *of size* 2^k *and a collision-resistant hash function* H*, we construct a full binary tree with* 2^k *leaves, where each leaf corresponds to an element of the vector. The value* v_n *of each node* n *in the tree is computed as follows:*

1. *If* n *is a leaf node, then* $v_n = H(\mathbf{v}[i])$*, where* i *is the index of the vector element corresponding to leaf* n*.*
2. *If* n *is an internal node with left child* L *and right child* R*, then* $v_n = H(v_L||v_R)$*.*

The digest d *of the Merkle tree accumulator is defined as the value of the root node.*

The Merkle tree supports the following operations (where how the auxiliary states stored in the prover changes are not shown)

- *Prove*$(i) \rightarrow \{n_1, n_2, \ldots, n_k\}$*: Generates a Merkle membership proof for the value* $\mathbf{v}_i$*. The proof includes the sibling values of all nodes along the path from the* i*-th leaf to the root.*
- *Verify*$(d, v_i, \{n_1, n_2, \ldots, n_k\}) \rightarrow \{0, 1\}$*: Verifies whether a value* v_i *belongs to the Merkle tree with root digest* d*. This is done by reconstructing the path to the root using the provided sibling hashes and checking if the recomputed root equals* d*. Verification requires* $O(k)$ *time.*
- *Update*$(i, v) \rightarrow d$*: Compute the hash* $h = H(v)$*, and recompute all hashes along the path from the updated leaf* i *to the root to produce the new digest* d*.*
- *UpdateH*$(i, h) \rightarrow d$*: Similar to Update, but instead of computing the hash*

internally, the caller provides the precomputed hash h of some value v. The function directly updates the hash stored at leaf i with h and recomputes the hashes along the path to the root.

2.6 Succinct Non-Interactive Arguments of Knowledge (SNARK)

Definition 6 (SNARK)**.** *For a binary relation R, A SNARK is a triple of PPT algorithms $\Pi = (Setup, Prove, Verify)$:*

- *$Setup(1^\lambda, R) \to (pk, vk)$: On input security parameter λ and the binary relation R, it outputs a common reference string consisting of the prover key and the verifier key (pk, vk).*
- *$Prove(pk, x, w) \to \pi$: On prover key pk, public input x and the witness w, it outputs a proof π.*
- *$Verify(vk, x, \pi) \to \{0, 1\}$: On verifier key vk, public input x, and a proof π, it outputs either 1 indicating $(x, w) \in R$ or 0 when $(x, w) \notin R$.*

The SNARK must satisfy completeness, knowledge soundness, non-interactivity, and succinctness. In some cases, it also provides zero knowledge, in which case it is referred to as a zk-SNARK. We refer readers to Appendix A for formal definitions.

2.6.1 R1CS

To utilize the SNARK for proving and verifying the binary relation R, we oftentimes encode the relation as an arithmetic circuit. An arithmetic circuit over a field F is a directed acyclic graph (DAG) where each vertex represents either a variable, a constant from F, or an arithmetic operation (addition or multiplication) over F. The edges represent the flow of values between vertices, with each edge carrying an element of F.

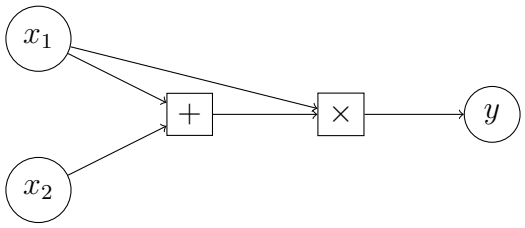


Figure 2.1: An example arithmetic circuit representing $(x_1 + x_2) \cdot x_1$

Definition 7 (Rank-1 Constraint System (R1CS))**.** *R1CS is a common format for representing arithmetic circuits. Formally, an R1CS over F with n variables (including inputs, witness variables, and outputs) and m constraints is defined by a tuple $(\mathbf{A}, \mathbf{B}, \mathbf{C})$, where:*

- $\mathbf{A}, \mathbf{B}, \mathbf{C}$ *are $m \times n$ matrices with entries in F*
- *The variables are represented by a vector $\mathbf{z} = (z_1, \ldots, z_n) \in F^n$, where $z_1 = 1$ is constant, $z_2, \ldots, z_k$ are public inputs and $z_{k+1}, \ldots, z_n$ are witness values.*

A vector $\mathbf{z}$ satisfies the R1CS if the following rank-1 quadratic equation holds:

$$(\mathbf{Az}) \cdot (\mathbf{Bz}) = \mathbf{Cz}$$

Relation between the arithmetic circuit and the R1CS. The R1CS representation allows us to encode the behaviour of an arithmetic circuit as a set of algebraic constraints. Each constraint ensures that a particular computation (such as an addition or multiplication) between variables is performed correctly. These variables include both public inputs and *witnesses* - the private intermediate values and auxiliary inputs known only to the prover.

For example, consider the statement: "I know some x such that $d = \texttt{sha256}(x)$." In this case, the witness includes the value x along with the intermediate computation traces produced during the evaluation of the hash function within the arithmetic circuit. Proving this statement (i.e., the correct evaluation of a `sha256` circuit) is equivalent to proving that the given public input (e.g., d) and the hidden witness (e.g., x and intermediate values) satisfy all the constraints that encode the `sha256` computation - ensuring that each intermediate step is correctly executed following `sha256` algorithms and that the final output matches d.

Therefore, as long as the verifier is convinced that the R1CS constraints accurately represent the intended arithmetic circuit and the resulting proof is valid, it can be confident that the relation, represented by the arithmetic circuit, is satisfied - without ever learning the witness itself.

`arkworks` [ac22] is a library that enables developers to express constraints as gadgets in Rust. Internally, it generates these constraints in the R1CS format, which can then be proved and verified with various SNARK implementations available within the ecosystem. In our work, we utilize `arkworks` to construct gadgets for proving BLS signature verification and committee rotation.

2.6.2 Recursive SNARK

A **recursive SNARK** is a SNARK that enables the composition of proofs. It allows a prover to generate a succinct proof for a statement that incorporates the verification of other proofs within the same system. Instead of proving $(x, w) \in R$ for some x, w, R, a recursive SNARK prover proves

$$\exists\ \Pi = (\text{Setup}, \text{Prove}, \text{Verify}), s.t., \text{Verify}(vk, x, \pi) = 1$$

Recursive SNARKs are particularly valuable in applications such as blockchain, where they enable the aggregation of state transitions into a single succinct proof, thereby reducing the prover's computational burden. For instance, recursive SNARKs can be employed to prove the evolution of a light client protocol by showing that, at round r:

- There exists a proof attesting to the validity of the checkpoint at round $(r-1)$, and
- The aggregated signature on the new checkpoint is a valid signature produced by a supermajority of the committee from round $(r-1)$.

2.6.3 Folding

A folding scheme is a technique that reduces multiple instances of the same relation into a single, compact instance while preserving the validity of the underlying statements.

Definition 8 (Folding scheme)**.** *(Adapted Definition from [NDC+24]) A folding scheme for a relation R is a tuple of PPT algorithms (Setup, Prove, Verify):*

- *Setup*$(1^\lambda, R) \to (fpk, fvk)$*: Given input security parameter and the relation R, it outputs a proving key fpk and a verifying key fvk.*
- *Prove*$(fpk, [(x_i, w_i)]_{i=1}^k) \to (x, w, \pi)$*: Given a proving key, and k instance-witness pairs claimed to be in R, it outputs a folded instance-witness pair (x, w) along with a folding proof π.*
- *Verify*$(fvk, [x_i]_{i=1}^k, x, \pi) \to \{0, 1\}$*: Given a verifying key, public input for k instances, a folded instance x, and a folding proof π, it verifies whether x is a folded instance of $[x_i]_{i=1}^k$ and outputs accept or reject.*

It should satisfy the following properties:

- *Completeness: If all initial instance-witness pairs are in the relation,* $[(x_i, w_i) \in R]_{i=1}^{k}$*, then it holds that the folding verifier will accept and the folded instance-witness pair also belongs to* R*,* $(x, w) \in R$*.*
- *Knowledge soundness: If an adversary* A *produces folded instances* $(x_i)_i^k = 1$*,* (x, w) *and folding proof* π *that are accepted by the verifier, and* $(x, w) \in R$ *then it must be the case that (except with negligible probability), an extractor can find witnesses* $[w_i]_{i=1}^{k}$ *such that* $[(x_i, w_i) \in R]_{i=1}^{k}$*.*

2.6.4 Incrementally Verifiable Computation (IVC)

IVC [Val08] enables verifiable computation for repeated function application. Intuitively, for a polynomial-time function F, with initial input (state) z_0 and witness values $\mathbf{w} = \{w_0, w_1, \ldots, w_{i-1}\}$, an IVC scheme allows a prover to produce a proof π_i for the statement

$$z_i = \underbrace{F(F(\ldots F(z_0, w_0), w_1) \ldots, w_{i-1})}_{i} = F^{(i)}(z_0, \mathbf{w})$$

In other words, it proves that z_i is the result of applying the function F iteratively i times to the initial value z_0, using the witness values $\mathbf{w}$.

Definition 9 (IVC)**.** *(Adapted Definition 5 from [KST22]) An IVC scheme for a polynomial-time function* F *is a tuple of PPT algorithms (Setup, Keygen, Prove, Verify):*

- *Setup*$(1^\lambda) \to pp$*: Given input security parameter, it outputs public parameters.*
- *Keygen*$(pp, F) \to (pk, vk)$*: Given input public parameters and the function* F*, it deterministic outputs a proving key* pk *and a verifying key* vk*.*
- *Prove*$(pk, (i, z_0, z_i), w_i, \Pi_i) \to \Pi_{i+1}$*: Given the proving key* pk*, public inputs* (i, z_0, z_i)*, witness* w_i*, and proof* Π_i *of* $z_i = F^{(i)}(z_0, \mathbf{w})$ *for some* $\mathbf{w}$*, it outputs a new proof* Π_{i+1}*.*
- *Verify*$(vk, (i, z_0, z_i), \Pi_i) \to \{0, 1\}$*: Given the verifying key* vk*, public inputs* (i, z_0, z_i) *and the IVC proof* Π_i*, it outputs 1 indicating* $z_i = F^{(i)}(z_0, \mathbf{w})$ *for some witness values* $\mathbf{w}$ *and 0 otherwise.*

Likewise, it needs to satisfy completeness and knowledge soundness.

- *Completeness:* $\forall \lambda \in N$, *for any PPT adversary* A*:*

$$\Pr\left[\begin{array}{l} \mathit{Verify}(vk, (i, z_0, z_i), \Pi_i) = 1 \\ \\ \\ \\ \end{array} : \begin{array}{c} pp \leftarrow \mathit{Setup}(1^\lambda), \\ F, \begin{pmatrix} i, z_0, z_i, z_{i-1}, \\ w_{i-1}, \Pi_{i-1} \end{pmatrix} \leftarrow A(pp) \\ pk,\ vk \leftarrow \mathit{Keygen}(pp, F), \\ z_i = F(z_{i-1}, w_{i-1}), \\ \mathit{Verify}(vk, (i, z_0, z_{i-1}), \Pi_{i-1}) = 1, \\ \Pi_i \leftarrow \mathit{Prove}\begin{pmatrix} pk, (i, z_0, z_{i-1}), \\ w_{i-1}, \Pi_{i-1} \end{pmatrix} \end{array}\right] = 1$$

  where F *is a polynomial time computable function.*

- *Knowledge Soundness:* $\forall \lambda \in N, n \in N$*, and any expected polynomial time adversaries* A*, there exists an expected polynomial time extractor* E *such that*

$$\Pr_r\left[\begin{array}{c} z_{i+1} \leftarrow F(z_i, w_i) \\ \forall i \in [0, n-1] \end{array} : \begin{array}{c} pp \leftarrow \mathit{Setup}(1^\lambda) \\ (F, (z_0, z), \Pi) \leftarrow A(pp, r) \\ \{w_0, \ldots, w_{n-1}\} \leftarrow E(pp, r) \end{array}\right] \approx$$
$$\Pr_r\left[\mathit{Verify}(vk, (n, z_0, z), \Pi) = 1 \ : \begin{array}{c} pp \leftarrow \mathit{Setup}(1^\lambda) \\ (F, (z_0, z), \Pi) \leftarrow A(pp, r) \\ (pk, vk) \leftarrow \mathit{Keygen}(pp, F) \end{array}\right]$$

  where r *denotes an arbitrarily long random tape.*

Nova [KST22] proposes how to use a folding scheme to construct an IVC scheme.

2.6.5 Folding-based SNARK for IVC

A **folding-based SNARK** is a SNARK that utilizes the IVC (folding) scheme to prove facts about a polynomial-time function F. Intuitively, it runs the IVC scheme to fold (prove) each step of the function F and compresses the final IVC proof π with a conventional SNARK.

Definition 10 (Folding-based SNARK)**.** *A folding-based SNARK for a polynomial-time function* F *is a tuple of PPT algorithms (Setup, Prove, Verify):*

- $\mathit{Setup}(1^\lambda, F) \rightarrow (pk, vk)$*: Given input security parameter and the function* F*,*

it outputs a proving key pk and a verifying key vk, in which

$$\begin{aligned} spk, svk &\leftarrow SNARK.Setup(1^\lambda, R) \\ pp &\leftarrow IVC.Setup(1^\lambda) \\ ipk, ivk &\leftarrow IVC.Keygen(pp, F) \\ pk &\leftarrow (spk, ipk) \\ vk &\leftarrow (svk, ivk) \end{aligned}$$

where R is a relation that checks there exists IVC proofs to verify state transitions.

- *$Prove(pk, (k, z_0, z_k), [(z_i, w_i)]_{i=1}^k) \rightarrow \pi$: Given the proving key, public inputs (k, z_0, z_k) and k input-witness pairs for F as witnesses, it outputs a proof π. The proof π is generated by using IVC.Prove and SNARK.Prove internally, and it contains a SNARK proof and some commitments about the (relaxed) R1CS instantiation of the function F we are proving.*
- *$Verify(vk, (k, z_0, z_k), \pi) \rightarrow \{0, 1\}$: Given a verifying key, public input (k, z_0, z_k), and a proof π, it verifies whether z_k is a state after applying F iteratively k times.*

As a SNARK, the folding-based SNARK also satisfies the standard properties of completeness and knowledge soundness. Intuitively, this follows from the corresponding guarantees provided by both traditional SNARKs and IVC. For details on the construction and formal proofs of these properties in the context of folding-based SNARKs, we refer readers to [KST22].

It is important to note that the IVC proving (folding) process is significantly faster than the final SNARK compression (Section 1.3 of [KST22]). This performance asymmetry suggests that the compression proof should be generated only when necessary, allowing for more efficient use of computational resources.

`sonobe` [pse23] is a library that implements a wide range of folding schemes to support the folding of arithmetic circuits in an IVC style, within the `arkworks` ecosystem. It allows developers to define circuits using `arkworks` and generate folding-based SNARK proofs from them. In our work, we utilize `sonobe` to construct SNARKs for verifying committee rotation.

Chapter 3

Related Work

We now overview existing cryptographic techniques that people use to try to speed up light client protocols and blockchain. For a more broad range of techniques that people apply to speed up light client protocol, we refer readers to Section 3 of [AKMW24].

zkEVM. zk-SNARKs have gained significant traction for enabling compact, verifiable proofs of complex computations. In the context of zkEVM [LLM+24], they are used to prove the correctness of Ethereum Virtual Machine (EVM) transaction execution. zkEVM generates a cryptographic proof that validates state transitions resulting from EVM computations, without revealing underlying data. This is done by constructing a circuit that models the EVM's execution logic - including arithmetic operations, memory handling, and stack manipulations - which is then proven using a zk-SNARK system. The resulting proof is verified on-chain, offloading computation from validators and enhancing scalability. However, zkEVM is primarily concerned with proving transaction execution correctness and does not address light client synchronization. As such, it is orthogonal to our goal of optimizing light client protocols. Other related approaches, such as assembly-based ZK-VMs like Cairo and RISC0 [CRTA+24], share similar limitations and are likewise orthogonal to our focus.

zkBridge. The zkBridge protocol [XZC+22] leverages zk-SNARKs to enable secure and efficient cross-chain communication. Specifically, zkBridge generates zk-SNARK proofs of a source chain's state transitions - modelled using a light client protocol - which are then verified on a target chain, enabling trustless interoperability across blockchains. However, zkBridge is designed for cross-chain scenarios, where verification occurs on a different chain, whereas our work focuses on a single-chain

setting, where verification is performed off-chain by the light client. Additionally, the SNARK construction used in zkBridge to incrementally prove the correctness of state evolution relies on a naive recursive zk-SNARK, which has limitations: it can only process one transition at a time, or a small batch - where the latter incurs higher proving overhead and increased latency. These limitations underscore the need for alternative proof techniques better suited to efficient light client protocols in our context.

Mina. Unlike previous approaches, Mina [BMRS20a] adopts a fundamentally different strategy by building a blockchain entirely around SNARKs, ensuring that both consensus and transaction execution are cryptographically verifiable. This design enables light clients to achieve the same security guarantees as full nodes, as the entire blockchain state can be verified in constant time - approximately 200 ms with a proof size of just 864 bytes. This makes Mina particularly suitable for resource-constrained devices. The protocol leverages recursive SNARKs to construct a binary tree of proofs that efficiently compresses state transitions, attesting to the validity of the entire chain up to the latest block. Notably, the proof generation cost scales only with the number of new transactions since the last block, rather than the full chain history.

However, Mina's approach is not directly applicable to our setting due to key differences in goals and constraints. Our objective is to develop a generalized and flexible protocol for light clients that can operate across a broader range of blockchains, particularly those that already use checkpoints. Mina, by contrast, is purpose-built from the ground up to integrate SNARKs into every layer of its architecture. This tight integration allows Mina to maintain a constant-sized summary of the blockchain, but it does not address the central challenge we tackle: retrofitting a proof mechanism onto an existing chain with a potentially large number of historical checkpoints. In our case, each prior checkpoint must be equipped with a verifiable proof of validity, requiring an efficient way to bootstrap and scale the proof generation process.

Plumo. Aligned with our overarching goal, Plumo [VGS+22] aims to enhance light client protocols with cryptographic proofs, enabling resource-constrained devices, such as low-end mobile phones, to securely sync with the latest blockchain state without relying on centralized intermediaries. Its design relies on a simplifying assumption (SA) that a supermajority of validators in each epoch remain honest. Unlike our approach, Plumo does not aim to generate a compact proof for every checkpoint in the chain. Instead, it focuses on blockchain state summaries, where the proof size grows sublinearly by concatenating proofs of prior summaries to validate the current one.

Plumo achieves efficiency through two SNARK-friendly cryptographic constructions: a BLS-based offline aggregate multisignature scheme (BBSGLRY), and a composite hash function (BHP-BLAKE2s). BBSGLRY aggregates epoch-level multisignatures into a single signature, significantly reducing verification constraints compared to verifying individual signatures. Notably, it improves upon prior schemes such as AMSP-PoP by removing the requirement for signers to know the aggregate public key ahead of time. The BHP-BLAKE2s hash function merges the algebraic collision-resistant Bowe-Hopwood-Pedersen hash with the symmetric BLAKE2s hash, balancing SNARK efficiency (i.e., fewer multiplication gates) with strong security guarantees. To circumvent the inefficiencies of non-native arithmetic in SNARK circuits, Plumo adopts a two-chain pairing curve design: BLS12-377 for consensus and BW6-761 for proof generation and verification.

Despite its innovations, Plumo's design is not directly applicable to our setting. Its focus on producing succinct summaries results in proofs that grow - albeit sublinearly - over time, which makes it unsuitable for use cases like ours that require a compact proof per checkpoint. This growth can become burdensome in chains with frequent checkpoints. Additionally, Plumo optimizes specifically for signatures based on BLS12-377, which limits its generalizability to blockchains that use different elliptic curves. Nevertheless, some of Plumo's techniques - especially its use of aggregate multisignatures - could be adapted in our context to reduce verification costs by aggregating signatures across multiple checkpoints into a single proof.

Transitive Signatures. In contrast to prior approaches that rely on SNARKs, [AKMW24] proposes a novel light client design for permissionless blockchains that avoids the computational overhead associated with zero-knowledge proofs. The authors build their protocol on two empirical observations: (i) most light clients synchronize missed states within a short window - typically under two hours - and (ii) in many committee-based permissionless blockchains, large subsets of validators remain unchanged across epochs, with validator turnover often below 7% per epoch. By leveraging these patterns, the protocol significantly reduces verification complexity using transitive signatures. Instead of performing traditional sequential verification, which requires $O(m)$ time to verify m block headers, this method enables constant-time ($O(1)$) verification in the common case of a stable validator quorum. However, the protocol's efficiency hinges on validator set stability, making it less suitable for blockchains with frequent validator churn - such as Ethereum - where new committees are sampled regularly. This limits the generality and applicability of their approach in more dynamic environments.

Chapter 4

Design

In this chapter, we present our protocol for verifying committee rotations in a light client setting. We begin by outlining the abstract model of the light client protocol that forms the basis of our approach. We then describe the full protocol and argue its security properties. Finally, we introduce the **Levelled Merkle Forest**, a data structure designed to optimize protocol performance.

4.1 Abstraction of the Light Client Protocol

Before delving into the details of our protocol, we provide an architectural abstraction of the light client model on which our protocol is built.

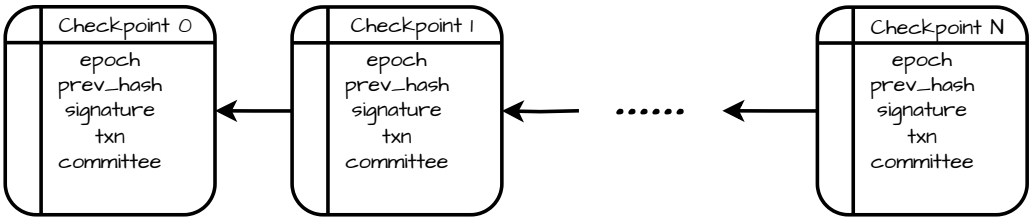


Figure 4.1: An abstraction of the chain of checkpoints in the light client protocol.

In this model, prover nodes (e.g., full nodes) maintain a chain $\mathcal{C}$ of *checkpoints* as shown in 4.1. Each checkpoint includes:

- An aggregated signature, accompanied by a map indicating which committee members from the previous checkpoint contributed to the aggregation;
- A cryptographic hash of the previous checkpoint;

- Transaction metadata; and
- The public keys of the next committee and their corresponding weights.

In this chain, the N-th checkpoint is signed by a supermajority of the committee from the $(N-1)$-th checkpoint. By signing the new checkpoint, the supermajority attests to its validity, thereby upholding the security property defined in Definition 1. The light client verifies the signature and checks that the sum of the weights associated with the aggregated public keys exceeds a strong quorum threshold. It is assumed that the committee at the genesis checkpoint (checkpoint 0) is trusted by all light clients. This trust stems from the security guarantees of the underlying blockchain protocol responsible for selecting the initial committee.

To access any transaction data it wants, the client requests the relevant information from the prover nodes, which provides proof to the client about the validity of the data it gives to the light client that the light client can verify based on the algorithm shown in Protocol 1.

Protocol 1 (Naive Light Client Protocol)**.** The naive light client protocol for a chain $\mathcal{C}$ with genesis checkpoint cp_0 is a tuple of algorithms (Setup, Request, Verify):

- $\text{Setup}(1^\lambda, \mathcal{C}) \to gen$: Given the input security parameter and checkpoint chain $\mathcal{C}$, the setup algorithm returns the genesis checkpoint $gen = cp_0$.
- $\text{Request}(i) \to (cp_i, (cp_0, cp_1, \ldots, cp_{i-1}))$: The light client requests cp_i. The prover nodes provide cp_i with proof about the inclusion of cp_i in the chain of checkpoints. The proof is simply a vector of checkpoints from the genesis checkpoint to the one before the light client requests.
- $\text{Verify}(gen, cp_i, (cp_0, cp_1, \ldots, cp_{i-1})) \to \{0, 1\}$: The light client verifies the proof by ensuring the genesis checkpoint is the one it trusts (checking $gen = cp_0$) and verify the committee rotation in the proof one by one (the supermajority of the committee in cp_k signs cp_{k+1} for $k \in [0, i-1]$). It outputs 1 if all the aforementioned verification is successful.

However, this naive light client protocol is not efficient as the size of the proof scales with the index of the checkpoint the light client requests. To address this problem, we propose the following protocol based on folding-based SNARK.

4.2 Protocol detail

Protocol 2 (Light Client Protocol with folding-based SNARK)**.** The light client protocol with folding-based SNARK for a chain $\mathcal{C}$ with genesis checkpoint cp_0 is a tuple of algorithms (Setup, Request, Verify):

- Setup$(1^\lambda, \mathcal{C}) \rightarrow (gen, pk, vk)$: Given the input security parameter and the checkpoint chain $\mathcal{C}$, the setup algorithm outputs the genesis checkpoint $gen = cp_0$ and the result of FoldingSNARK.Setup$(1^\lambda, F)$, where F verifies the committee rotation of $\mathcal{C}$. The function F is discussed in more detail in Section 4.2.2.
- Request$(pk, i) \rightarrow (cp_i, \pi)$: Given the proving key pk and the light client's request for ith checkpoint, the prover nodes provides cp_i with a proof.
- Verify$(vk, gen, cp_i, \pi) \rightarrow \{0, 1\}$: The light client verifies the proof by ensuring FoldingSNARK.Verify$(vk, (i, gen, cp_i), \pi) = 1$.

To implement this protocol, a crucial aspect is to model the function F. In the following sections, we present the core component for verifying committee rotation - the BLS verification circuit - and introduce the interface and implementation of the folding circuit $\mathcal{FOLD}$, which is the concrete realization of the function F.

4.2.1 BLS Verification Circuit

Circuit $\mathcal{BLS}$

Public input: cp_i, apk, sig;
Computation:;
(1) Check BLS.Verify$(sig, cp_i) = 1$;
(1.1) Compute the hash $h = H(cp_i)$, where H is a collision-resistant hash mapping cp_i to the bilinear group G_2, implemented in R1CS;
(1.2) Verify the pairing equation $e(apk, h) = e(g, sig)$, where g is the generator of the bilinear group G_1.

Figure 4.2: BLS verification circuit.

Figure 4.2 illustrates the BLS verification circuit. It takes in input checkpoint cp_i, aggregated public key apk, and a signature sig. It consists of two main steps: hashing the checkpoint to an element in the group G_2 and verifying a pairing equation. Since we employ BLS signatures over a BLS curve, we instantiate the hash function H

according to the construction in [WB19], which provides a deterministic, collision-resistant hash for mapping byte sequences to elements in G_2. This construction builds upon a base hash function $H' : \{0,1\}^* \to \{0,1\}^k$ and involves the following steps:

- **HashToField**(bytes) $\to$ elm: Given an input byte sequence, extract sufficient bits from H'(bytes) to construct a field element elm in the domain of G_2.
- **MapToCurve**(elm) $\to P$: Map the field element elm to a point P on the elliptic curve (not necessarily in the correct subgroup).
- **CofactorClearing**$(P) \to P'$: Clear the cofactor of P to obtain a point P' in the correct subgroup of G_2.

We discuss the challenges of concretely implementing these steps in the `arkworks` ecosystem in Section 5.2.

4.2.2 Folding Circuit

Let's recall the definition of the function F. It is a function that takes in two inputs z_i, w_i, which are the state and witness at ith step respectively and outputs a new state z_{i+1}.

To model this in R1CS, it's sufficient to define a circuit that takes in two public inputs (z_i, z_{i+1}) and a witness value w_i. We don't need to compute the state z_{i+1} in R1CS as the purpose of the R1CS is to *verify* the state transition rather than computing it. The computation is usually done out of the circuit.

Based on this interface, we define the folding circuit $\mathcal{FOLD}$ as follows:

Circuit $\mathcal{FOLD}$

Public input: (cp_i, cp_{i+1});
Witness: sig;
Computation:;
(1) Aggregate all the pks from cp_i that sign sig to apk;
(2) Check the sum of the weight of these pks exceeds the strong quorum threshold;
(3) Check the aggregated signature sig is a valid elliptic curve group element and is in the correct subgroup;
(4) Run $\mathcal{BLS}((cp_{i+1}, apk, sig),)$;

Figure 4.3: Folding circuit for proving committee rotation.

This circuit ensures that the aggregated public key corresponds to a supermajority of the committee (Step 2) and that the associated signature is verified correctly (Step 4). Additionally, it performs the necessary checks to confirm that the aggregated signature is a valid element of the elliptic curve group and lies within the correct subgroup (Step 3), thereby mitigating the risk of small subgroup attacks. These checks are essential, particularly because in the folding circuit $\mathcal{FOLD}$, the signature sig is provided as a witness.

4.2.3 Security Proof

In this section, we prove that Protocol 2, with the function F modelled as in Figure 4.3, satisfies soundness and completeness, which are defined as follows:

Definition 11. *The soundness and completeness of the light client protocol (with a proof system) are defined as follows:*

- ***Soundness**: $\forall \lambda \in N$, any checkpoint chain $\mathcal{C}$, and any PPT adversary A, the following probability is negligible with respect to λ:*

$$\Pr\left[\begin{array}{rl} & (gen, pk, vk) \leftarrow Setup(1^\lambda, \mathcal{C}) \\ Verify(vk, gen, cp, \pi) = 1 : & (cp, \pi) \leftarrow A((gen, pk, vk), \mathcal{C}), \\ & cp \notin \mathcal{C} \end{array}\right]$$

- ***Completeness**: $\forall \lambda \in N$, any checkpoint chain $\mathcal{C}$, and any $cp_i \in \mathcal{C}$*

$$\Pr\left[Verify(vk, gen, cp_i, Request(pk, i)) : \ (gen, pk, vk) \leftarrow Setup(1^\lambda, \mathcal{C})\right] = 1$$

The completeness of the protocol follows directly from the completeness of the underlying SNARK and IVC schemes. We now focus on proving soundness.

Proof. Suppose an adversary outputs a pair (cp, π) such that $\text{Verify}(vk, gen, cp, \pi) = 1$. By the knowledge soundness of folding-based SNARKs (which itself follows from the knowledge soundness of both SNARKs and the IVC scheme), we can extract a sequence of witness values $(cp'_1, \ldots, cp'_{n-1}, cp)$. Let us denote the corresponding checkpoint chain as $\mathcal{C}' = (cp'_0, cp'_1, \ldots, cp'_n)$, where $cp'_0 = gen$ and $cp'_n = cp$.

By the definition of the folding circuit in Figure 4.3, this chain $\mathcal{C}'$ satisfies the following property:

- For every $i \in [0, n-1]$, the supermajority of the committee at checkpoint cp'_i has signed the subsequent checkpoint cp'_{i+1}.

Now, consider the canonical chain $\mathcal{C} = (cp_0, \ldots, cp_n)$ maintained by the prover nodes, where $cp_0 = gen$. Without loss of generality, suppose $\mathcal{C}$ has the same length as $\mathcal{C}'$. Since $cp \notin \mathcal{C}$, the two chains must differ at some position. Let i be the smallest index at which $\mathcal{C}$ and $\mathcal{C}'$ diverge.

This implies that the supermajority of the committee at checkpoint $cp_{i-1} = cp'_{i-1}$ has signed two distinct checkpoints: cp_i and cp'_i. This contradicts the security assumption stated in Definition 1, which requires that the supermajority of a committee cannot sign conflicting checkpoints. Hence, soundness is preserved. □

4.3 Levelled Merkle Forest

4.3.1 The Bootstrapping Problem

A significant challenge in the above scheme is the need to generate a SNARK proof for every checkpoint to enable efficient verification by light clients. This requirement introduces substantial proving overhead, as SNARK compression is considerably more time-consuming than the folding process itself. As a result, it may take a long time for prover nodes to produce proofs for all existing checkpoints - particularly those that predate the deployment of our proving scheme. We refer to this challenge as the *bootstrapping problem*.

To alleviate this proving burden, we can trade off some proving efficiency for a slight increase in verification time for checkpoints that existed before our scheme was introduced. A straightforward solution is to prove the construction of a Merkle accumulator within the R1CS circuit, where the hash of each checkpoint is stored as a leaf in a Merkle tree. The number of leaves, n, is fixed in advance and can be set to match the number of pre-existing checkpoints.

Under this design, the system produces a SNARK compression proof only once every n checkpoint. Then, for any specific checkpoint cp_i where $i \in [1, n)$, the prover supplies a proof tied to the n-th checkpoint. This proof includes the Merkle root $root$ and a Merkle path demonstrating that cp_i is a valid leaf in the Merkle tree rooted at $root$. The light client first verifies the validity of the Merkle root using SNARK and then verifies cp_i using the Merkle root. As a result, the light client's verification time increases from $O(1)$ to $O(\log n)$. The corresponding prove and verify algorithms are summarized in Protocol 3.

Protocol 3 (Light Client Protocol with folding-based SNARK + Merkle Root)**.** The light client protocol with folding-based SNARK + Merkle Root for a chain $\mathcal{C}$

with genesis checkpoint cp_0 is a tuple of algorithms (Request, Verify):

- Setup$(1^\lambda, \mathcal{C}) \rightarrow (gen, pk, vk)$: Given the input security parameter and the checkpoint chain $\mathcal{C}$, the setup algorithm outputs the genesis states $gen = (cp_0, root_0)$, where $root_0$ is the Merkle digest of accumulating a vector of zero hashes, and the result of FoldingSNARK.Setup$(1^\lambda, F)$.
- Request$(pk, i) \rightarrow (cp_i, (cp_n, root_n, path, \pi))$: Given the proving key pk and the light client's request for ith checkpoint, the prover nodes provides cp_i with a proof. The proof contains cp_n (the nth checkpoint), $root_n$ (the Merkle root at the nth step), a folding-based SNARK proof π and a Merkle path $path$.
- Verify$(vk, gen, cp_i, (cp_n, root_n, \pi)) \rightarrow \{0, 1\}$: The light client verifies the proof by ensuring FoldingSNARK.Verify$(vk, (n, gen, (cp_n, root_n)), \pi) = 1$ and Merkle.Verify$(root_n, cp_i, path) = 1$.

Security. The security of Protocol 3 follows from the security of Protocol 2 and the security (binding) of the Merkle accumulator under the assumption that the hash function used is collision-resistant.

We implement this idea by augmenting the circuit described in Figure 4.3, as shown below:

Circuit $\mathcal{FOLDM}_{naive}$

Public input: $((cp_i, root_i), (cp_{i+1}, root_{i+1}))$;
Witness: $(sig, path_i)$;
Computation:;
(1) Run $\mathcal{FOLD}((cp_i, cp_{i+1}), sig)$;
(2) Check Merkle.Verify$(root_i, 0, path_i) = 1$;
(3) Check Merkle.Verify$(root_{i+1}, H(cp_i), path_i) = 1$;

Figure 4.4: Folding circuit with Merkle root for proving committee rotation.

To support the Merkle accumulator, we augment the circuit with two Merkle roots as public inputs: one representing the current tree root and the other representing the updated root after inserting the hash of a new checkpoint. To verify correct insertion, the circuit ensures that the insertion position initially contained an empty value (Step 2) and that replacing it with the checkpoint's hash produces the new root (Step 3). These checks require a Merkle path witness of size $O(\log n)$ and $O(\log n)$ hash computations.

However, a key limitation of the naive Merkle accumulator is that it treats all past checkpoints equally. In practice, as observed in [AKMW24], light clients most frequently synchronize with the most recent checkpoint. Motivated by this observation, we hypothesize that recent checkpoints are accessed far more often than older ones. To exploit this skewed access pattern, we introduce a new data structure: the **Levelled Merkle Forest (LMF)**. This structure yields variable-length proofs - offering shorter proofs when the queried checkpoint index i is close to n - while still maintaining a worst-case proof size of $O(\log n)$.

4.3.2 Construction

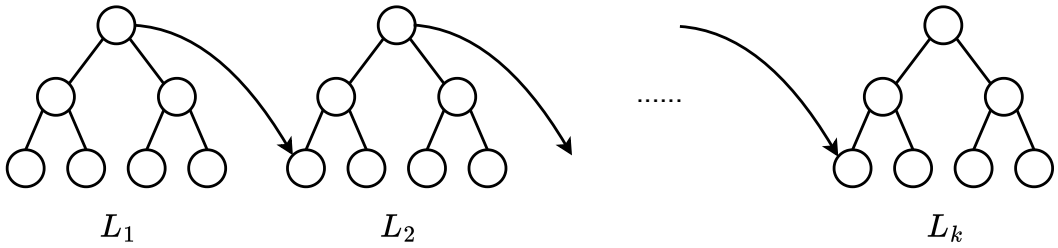


Figure 4.5: The Levelled Merkle Forest.

Figure 4.5 illustrates the structure of the **Levelled Merkle Forest** (LMF). The forest consists of k complete Merkle trees, $M_1, M_2, \ldots, M_k$, each assigned to a distinct level $L_1, L_2, \ldots, L_k$, and each containing the same number of nodes, q. A defining structural property of the LMF is that the root of the Merkle tree at level L_i becomes a leaf node in the Merkle tree at the next higher level, L_{i+1}. As a result, only the leaves of the first-level tree, M_1, directly store hashes of external data.

Figure 4.6 describes the off-circuit algorithms to operate on the LMF.

The main idea of the **Construct-Naive** algorithm is to incrementally update the state s while ensuring that all previously constructed Merkle trees remain accessible for indexing in the future. This is achieved by storing them in a map, with careful attention to how indexing is managed. A key observation is that each Merkle tree in the Levelled Merkle Forest (LMF) is a complete binary tree, and the LMF is built hierarchically. As a result, different Merkle trees in the forest use different segments of the bit representation of the index in the vector $\mathbf{v}$.

For example, the lowest $\lfloor \log_2 q \rfloor$ bits (from the least significant bit) can be used to index into M_1, while the remaining higher-order bits are used as a key in $\mathbf{m}_1$ to identify and store this specific instance of M_1. Also, as can be seen from the

Parameters Descriptions:

- H: A collision-resistant hash function with a desired security level.
- n: The number of nodes that can be stored in the LMF.
- q: The number of nodes in each tree.
- k: The number of trees in LMF.
- $\mathbf{s}$: The LMF as shown above in Figure 4.5. $\mathbf{s} = (M_1, \ldots, M_k)$.
- $\mathbf{m}$: A vector of k maps (one for each level) from index to Merkle tree.
- d: The digest of the LMF tree.
- $\mathbf{v}$: The input vector of length n.

Construct-Naive($\mathbf{v}$) $\rightarrow \mathbf{s}, \mathbf{m}, d$:

- Initialize an empty LMF state $\mathbf{s}$ and an empty map $\mathbf{m}$.
- From $i = 0$ to n (exclusive),
  - $h = H(\mathbf{v}_i)$
  - $idx = i$
  - From $j = 1$ to k
    - idx_j = lower $\lfloor \log_2 q \rfloor$ bits of idx.
    - M_j.UpdateH(idx_j, h).
    - Right shift idx by $\lfloor \log_2 q \rfloor$ bits.
    - Store M_j to $\mathbf{m}_j$ with idx as key.
    - h = root of M_j.
- Return $\mathbf{s}, \mathbf{m}, d$ = root of M_k.

Construct-Fast($\mathbf{v}$) $\rightarrow \mathbf{s}, \mathbf{m}, d$:

- Initialize an empty map $\mathbf{m}$.
- From $i = 1$ to k,
  - If $i = 1$, $\mathbf{m}_i$ = trees in L_i init using $\mathbf{v}$;
  - Else, $\mathbf{m}_i$ = trees in L_i init using the digest of trees in L_{i-1}.
- $\mathbf{s}$ = vector of last trees constructed in each level.
- Return $\mathbf{s}, \mathbf{m}, d$ = root of M_k.

Prove($\mathbf{s}, \mathbf{m}, i$) $\rightarrow$ path:

- **path** = an empty vector
- From $j = 1$ to k
  - idx_j = lower $\lfloor \log_2 q \rfloor$ bits of i.
  - Right shift i by $\lfloor \log_2 q \rfloor$ bits.
  - M = the Merkle tree stored at key i in $\mathbf{m}_j$.
  - proof = M.Prove(idx_j, h), and add proof to **path**.
- Return **path**.

Verify(d, v_i, path) $\rightarrow \{0, 1\}$: Follows the standard Merkle tree verification process: first, compute the hash of v_i; then, iteratively hash it with the sibling values in **path** to reconstruct the root d'; finally, return if $d = d'$.

Figure 4.6: The off-circuit algorithm for Levelled Merkle Forest.

Construct-Naive algorithm, the LMF can be built incrementally - by inserting elements one at a time.

However, this naive approach is not ideal, as it incurs a time complexity of $O(Nk \log q)$: there are N values in the vector, and each value must update k Merkle trees (each of size q). To improve this, we can leverage the same idea used in constructing traditional Merkle trees and binary heaps. Specifically, we first construct all the trees at level L_1 and store them in a map. Using these, we can efficiently construct the trees at level L_2, and so on. This strategy ensures that each node in all the Merkle trees is assigned a value only once, leading to a reduced time complexity. We refer to this optimized method as the **Construct-Fast** function.

The **Prove** algorithm uses a similar strategy for index manipulation. It splits the index i into two parts: the lower-order bits determine the position of the element within a specific Merkle tree and are used to call the Merkle tree's **Prove** API, while the higher-order bits are used to look up the correct Merkle tree from the map. The final LMF proof is a concatenation of k individual Merkle proofs, which together allow reconstruction of the overall digest of the LMF.

Unlike the off-circuit algorithm, the on-circuit algorithm differs in that it does not need to maintain a map. Instead, it is only responsible for verifying that the state transitions of the LMF are valid. This is achieved by executing a modified version of the **Construct-Naive** algorithm within the circuit, which verifies the LMF state step-by-step. This incremental verification is made possible by the folding-based SNARK, which ensures the correctness of the entire construction process. The corresponding circuit is illustrated in Figure 4.7.

Circuit $\mathcal{LMF}$

Public input: (s_i, s_{i+1});

Witness: $(index, hash)$;

Computation:;

(1) Check $index = i$;

(2) $h = hash$

From $j = 1$ to k

- idx = lower $\lfloor \log_2 q \rfloor$ bits of $index$.
- M_j.UpdateH($idx, hash$).
  - Rehash all the states with the new hash h to compute the new root.
- Right shift $index$ by $\lfloor \log_2 q \rfloor$ bits.
- Update h = root of M_j.

Figure 4.7: R1CS circuit to verify the LMF construction (one step).

In the R1CS setting, the Merkle.UpdateH function must be adapted because the *index* is a witness value, which means its value is not known at circuit synthesis time. As a result, the circuit cannot perform selective updates based on the index. The only viable solution is to recompute all the hashes to derive the Merkle root from scratch. This implies that the cost of verifying a state change within the R1CS scales linearly with the size of the state input s_i embedded in the circuit.

Batch Update. As evident from the design of the $\mathcal{LMF}$ circuit, the entire state must be rehashed for each update. This observation naturally motivates the idea of batching multiple updates within the circuit. Unlike traditional Merkle accumulators - where batching updates entail applying each update sequentially and thus incur the number of constraints proportional to the batch size - the $\mathcal{LMF}$ circuit rehashes the entire state regardless of the number of updates. As a result, batching updates can significantly reduce amortized proving time. Figure 4.8 presents the batch update circuit for LMF.

Circuit $\mathcal{LMF}_{batch}$

Public input: (s_i, s_{i+q}) where i is a multiple of q;

Witness: $(index, \{hash_1, \ldots, hash_q\})$;

Computation:;

(1) Check $index = i$;

(2) M_1.UpdateH$(index, \{hash_1, \ldots, hash_q\})$

- Recompute the digest for M_1 by using $\{hash_1, \ldots, hash_q\}$ as leaves.

$h =$ root of M_1.

From $j = 2$ to k

- Right shift $index$ by $\lfloor \log_2 q \rfloor$ bits.
- $idx =$ lower $\lfloor \log_2 q \rfloor$ bits of $index$.
- M_j.UpdateH$(idx, hash)$
- Update $h =$ root of M_j.

Figure 4.8: R1CS circuit to verify the LMF construction (batched).

4.3.3 On-Circuit and Off-Circuit Cost Analysis

In this section, we examine the parameterization of the LMF to identify optimal values for specific objectives.

Consider a vector of length N that we wish to commit to, and suppose each Merkle tree has q nodes. We observe the following:

- The number of level-L_1 trees required is $\frac{N}{\frac{q+1}{2}} = \frac{2N}{q+1} < \frac{2N}{q}$.
- Similarly, the number of level-L_2 trees is $\frac{N}{\left(\frac{q+1}{2}\right)^2} = \frac{4N}{(q+1)^2} < \frac{4N}{q^2}$.

This suggests that the number of Merkle trees required at level k can be upper-bounded by:

$$\frac{2^k N}{q^k}.$$

To determine the maximum number of levels k necessary for the LMF to store N values, we solve:

$$\frac{2^k N}{q^k} = 1 \quad \Rightarrow \quad k = \frac{\log N}{\log \frac{q}{2}}.$$

Given the practical use cases of LMF, we now consider two key objectives for optimization:

- **Total On-Circuit State Size**: The circuit must maintain k Merkle trees, each with q nodes, resulting in a total state size of:

$$kq = \frac{\log N}{\log \frac{q}{2}} \cdot q.$$

This metric is critical because it directly impacts the number of constraints generated in the R1CS representation, and hence the performance of the underlying SNARK.

  - **Number of Constraints**: Since verifying an LMF update on-circuit involves recomputing all hashes, the constraint count grows linearly with the number of states, i.e., $O(N)$.
  - **SNARK Performance**: While performance depends on the specific proving system - some like HyperPlonk have linear complexity [CBBZ23], while others like Groth16 is quasi-linear [Gro16] - in both cases, performance correlates closely with the number of constraints. Therefore, minimizing on-circuit state size is an important design goal.

- **Proof Size**: Based on Figure 4.6, the proof size is given by:

$$k \log q = \frac{\log N}{\log \frac{q}{2}} \cdot \log q.$$

This reflects the concatenation of k Merkle proofs, each of length $\log q$, and thus serves as another key metric.

Since both $\log \frac{q}{2}$ and $\log q$ are in $O(\log q)$, the proof size remains asymptotically close to $\log N$, regardless of the specific choice of q. Therefore, we choose to focus our optimization on minimizing the total on-circuit state size. This corresponds to minimizing the following function with respect to q:

$$\text{Minimize} \quad \frac{\log N}{\log \frac{q}{2}} \cdot q.$$

Taking the derivative and solving yields the optimal value:

$$q = 2e.$$

This suggests that, in theory, setting q close to $2e \approx 5.44$ minimizes the total on-circuit state size. In practice, since $q + 1$ must be a power of two to maintain the complete binary Merkle tree structure, we select the nearest such value, like $q = 3$ or $q = 7$, depending on the implementation trade-offs.

With the above optimization results, Figure 4.9 summarizes the bounds for all the operations of LMF.

The number of trees k: $\frac{\log N}{\log \frac{q}{2}} = O(\log N)$.
Off-Circuit:

- **State Size (including the map)**: As each level l has $\frac{2^l N}{q^l}$ Merkle trees, we can compute the total state size needed as

$$q \sum_{i=1}^{k} \frac{2^i N}{q^i} = qN(\frac{1 - (\frac{2}{q})^{k+1}}{1 - \frac{2}{q}} - 1)$$
$$< qN(\frac{q}{q-2} - 1) = \frac{2q}{q-2}N \qquad \text{(as q = 2e)}$$

  So, the state size needed is $O(N)$.
- **Construction Time (Fast)**: $O(N)$, since the state size is $O(N)$ and each node in the trees that make up the state is assigned a value exactly once.
- **Proof Size**: $k \log q = O(\log N)$.
- **Verification Time**: Because the proof is $O(\log N)$. verification also takes $O(\log N)$.

On-Circuit:

- **State Size**: $kq = 2e \ln N = O(\log N)$.

Figure 4.9: The on-circuit and off-circuit cost of LMF.

4.3.4 Variable Length Proof

At first glance, compared with the traditional Merkle accumulator, the LMF does not provide benefits - despite increasing the size of the public input from $O(1)$ to $O(\log N)$ - and also introduces additional complexity in constructing the accumulator and generating the proof.

However, for the same reason, the LMF provides a unique benefit when running in R1CS: it results in a state of $O(\log N)$ rather than $O(1)$. One observation about these states is that, given a value $\mathbf{v}_i$, if $n - i \leq \frac{q+1}{2}$ (where $\frac{q+1}{2}$ denotes the number of leaves in the tree and n is a power of the number of leaves), its hash is stored directly in M_1 of the state. Similarly, if $n - i \leq (\frac{q+1}{2})^2$, the hash of the M_1 that stores $\mathbf{v}_i$ is in the M_2 of the state. This applies generally: for any $\mathbf{v}_i$ such that $n - i \leq (\frac{q+1}{2})^l$, the hash of its M_{l-1} is stored in the M_l in the state. This suggests that, in proving, we don't need to provide a full Merkle path, but a path that is enough to reach a hash that is stored inside the final state.

Based on this observation, Figure 4.10 describes the variable length prove and verify

algorithm.

Parameters Descriptions:

- H: A collision-resistant hash function with a desired security level.
- n: The number of nodes that can be stored in the LMF.
- q: The number of nodes in each tree.
- k: The number of trees in LMF.
- nl: The number of leaves in each tree.
- $\mathbf{s}$: The LMF shown above in Figure 4.5. $\mathbf{s} = (M_1, \ldots, M_k)$.
- $\mathbf{m}$: A vector of maps from index to Merkle tree. The vector contains a map for each level k.
- d: The digest of the LMF tree.

VarProve($\mathbf{s}, \mathbf{m}, i$) $\rightarrow$ path:

- **path** = an empty vector.
- n = round n to next power of nl.
- diff = $\max(n - i - 1, 1)$
- $l = \lfloor \log_{nl} \text{diff} \rfloor + 1$
- From $j = 1$ to l (exclusive)
  - idx_j = lower $\lfloor \log_2 q \rfloor$ bits of idx.
  - Right shift idx by $\lfloor \log_2 q \rfloor$ bits.
  - M = the Merkle tree stored at key idx in $\mathbf{m}_j$.
  - proof = M.Prove(idx_j, h).
  - Add proof to **path**.

  } This part is identical to the corresponding part in **Prove**.
- Return **path**.

VarVerify($\mathbf{s}, v_i$, path) $\rightarrow \{0, 1\}$:

- diff = $\max(n - i - 1, 1)$
- $l = \lfloor \log_{nl} \text{diff} \rfloor$
- Follows the standard Merkle tree verification process: first, compute the hash of v_i; then, iteratively hash it with the sibling values in **path** to reconstruct the final digest; finally, return 1 if the reconstructed digest is in the corresponding leaf of $M_l \in \mathbf{s}$, and 0 otherwise.

Figure 4.10: The variable length prove and verify algorithm for Levelled Merkle Forest.

4.3.5 Light Client Protocol with LMF

With the LMF circuit in and the variable-length algorithms in Figure 4.10, we can define our final version of the light client protocol as below.

Protocol 4 (Light Client Protocol with folding-based SNARK + LMF)**.** The light

client protocol with folding-based SNARK + LMF for a chain $\mathcal{C}$ with genesis checkpoint cp_0 is a tuple of algorithms (Request, Verify):

- Setup$(1^\lambda, \mathcal{C}) \rightarrow (gen, pk, vk)$: Given the input security parameter and the checkpoint chain $\mathcal{C}$, the setup algorithm outputs the genesis states $gen = (cp_0, s)$, where s is the empty LMF state, and the result of FoldingSNARK.Setup$(1^\lambda, F)$.
- Request$(pk, i) \rightarrow (cp_i, (cp_n, s_n, mp, \pi))$: Given the proving key pk and the light client's request for ith checkpoint, the prover nodes provides cp_i with a proof. The proof contains cp_n (the nth checkpoint), s_n (the LMF state at the nth step), a folding-based SNARK proof π and a variable-length LMF proof mp.
- Verify$(vk, gen, cp_i, (cp_n, s_n, mp, \pi)) \rightarrow \{0, 1\}$: The light client verifies the proof by ensuring FoldingSNARK.Verify$(vk, (n, gen, (cp_n, s_n)), \pi) = 1$ and LMF.VarVerify$(s_n, cp_i, path) = 1$.

Security. Similar to Protocol 3, the security of Protocol 4 follows from the security of Protocol 2 and the security (binding) of the Merkle accumulator under the assumption that the hash function used is collision-resistant.

Chapter 5

Implementation

In this chapter, we describe the overall architecture of our library `Mim`, which implements our light client protocol. We then delve into specific components of the implementation, focusing on the implementation of hash to the curve, a fix for a bug related to emulated field variables in `arkworks`, and various circuit optimizations.

5.1 Architecture

Our library consists of 8000+ lines of Rust code and has several modules, described below:

- `bc`: Provides an implementation of the abstracted checkpoint chain described in Figure 4.1. It defines data structures representing signatures, committees, checkpoints, and the checkpoint chain, along with the necessary methods for verification and serialization. To facilitate testing, it also includes helper methods for randomly generating checkpoint chains.
- `bls`: Implements BLS signatures both on-circuit and off-circuit. It provides a convenient API, including `sign`, `aggregate_sign`, `verify`, and `aggregate_verify`. Both versions are generic over the BLS12 family of curves (e.g., BLS12-377 or BLS12-381) to facilitate easy switching of curves. Additionally, the on-circuit implementation is generic over the SNARK field and the field variable it uses (native or emulated).
- `hash`: Implements the hash-to-curve [WB19] in R1CS for BLS12 curves.

- `merkle`: Implements the LMF data structure and corresponding algorithms, both on-circuit and off-circuit, as described in the paper.
- `folding`: Uses `sonobe` to implement the folding circuit for verifying committee rotation on the abstracted checkpoint chain. To conform to `sonobe`'s interface, it introduces traits for converting between constraint field variables and high-level objects, as well as for serializing and deserializing these objects to bytes, which are consistent with the `bc` module. This module builds on top of `bc` and `bls`.
- `tests`: Contains tests and debugging notes that uncovered the emulated field variable bug in `arkworks`. It also investigates properties of emulated field variables, motivating the discussion in Section 7.3.

We will now highlight some aspects of `hash`, `bls`, and `folding` and describe how we uncover and fix the bug in `arkworks` in the following sections.

5.2 Hash to Curve

In this subsection, we detail how we implement the three parts of hash-to-curve: hash-to-field, map-to-curve, and cofactor clearing.

5.2.1 Hash to Field

The hash-to-field step consists of two parts: an *expander*, which expands the input to a sequence of uniformly random bytes using a cryptographic hash function, and a *hasher*, which reconstructs field elements from those bytes.

Expander

To implement the expander, we first define an interface for the hash function. Inspired by the `arkworks` and `rand` crates in Rust, we introduce a `PRFGadget` trait that abstracts the behaviour of the hash function. Specifically, the hash function should support incremental updates and a finalization step that outputs the digest.

```
pub trait PRFGadget<F: Field> {
    type OutputVar: EqGadget<F> + ToBytesGadget<F> + Clone + Debug;

    // Output size of the hash function in bytes
    const OUTPUT_SIZE: usize;
```

```
    fn update(
        &mut self, input: &[UInt8<F>]
    ) -> Result<(), SynthesisError>;

    fn finalize(self) -> Result<Self::OutputVar, SynthesisError>;
}
```

To realize this interface, we modified the existing R1CS implementation of the Blake2s hash function in `ark-crypto-primitives`. As the base implementation was non-incremental - i.e., it only accepted the entire input at once - we studied the standard Blake2s and implemented an incremental update mechanism that achieves a 1-to-1 match with the standard Blake2s output across our test suite.

Hasher

With the expander in place, the next step is to implement a hasher. The hasher requests a sufficient number of uniform bytes from the expander and transforms them into the target field element (often called a *field variable* in R1CS). To do this, we need to know the extension degree m of the field and the size s of a base prime field element. Using these, we compute the total number of bits needed to construct a field element as $m \cdot s$. As for what needs to be hashed, we follow the IRTF RFC 9380 [FHSS+23].

The main challenge lies in reconstructing the base prime field variable from bits and then building the field variable from its base field representation. Since the `arkworks` ecosystem does not provide utilities for these, we implemented our own.

To address the first challenge, we define a trait called `FromBitsGadget` for prime field variables. This trait provides a method to construct a prime field variable from bits. To ensure correctness and soundness, we follow the implementation of `Boolean::le_bits_to_fp` and apply a double-and-add algorithm inside R1CS.

```
pub trait FromBitsGadget<CF: PrimeField>: Sized {
    fn from_le_bits(bits: &[Boolean<CF>]) -> Self;
}
```

Finally, to reconstruct the full field element from base prime field elements, we define another trait, `FromBaseFieldVarGadget`. This trait is implemented for various field variables in `arkworks` and enables us to construct field elements from an iterator over their base field components.

```
pub trait FromBaseFieldVarGadget<CF: PrimeField>: Sized {
```

```
    type BasePrimeFieldVar: FromBaseFieldVarGadget<CF>
                            + FromBitsGadget<CF>;

    fn from_base_field_var(
        iter: impl Iterator<Item = Self::BasePrimeFieldVar>,
    ) -> Result<Self, SynthesisError>;
}
```

5.2.2 Map to Curve

With the hasher in place, the next step is to map the resulting field element to a point on the elliptic curve. This process involves two mappings: the simplified SWU (Shallue-van de Woestijne-Ulas) map, which deterministically maps a field element to a valid point on an isogenous elliptic curve, and the WB map, which translates points from this isogenous curve back to the target curve [WB19]. Since the WB map is essentially an isogeny evaluation - i.e., applying a predefined rational map via polynomial evaluation - we omit its details and instead focus on the SWU map.

To implement the map-to-curve, we follow the R1CS-based implementation of the off-circuit `arkworks` version of the map-to-curve algorithm. A major challenge here stems from the lack of native support for certain mathematical operations in the R1CS ecosystem - particularly the square root operation, which is not provided out of the box.

To address this, we use a well-known trick from circuit design in R1CS [Hou23]: rather than computing the square root directly (e.g., using the Tonelli-Shanks algorithm), we treat the square root as a witness and enforce its correctness via constraints. Specifically, we enforce that the square of the witness, if it exists, equals the input element.

We also draw from insights in [DHMP23], which explores the soundness pitfalls of on-circuit square root computation for `decaf377`. Based on that, we construct a circuit-safe square root gadget using the Legendre symbol to check whether the input element is a quadratic residue:

```
impl<F: PrimeField> SqrtGadget<F, F> for FpVar<F>{
    fn sqrt(&self) -> Result<(Boolean<F>, Self), SynthesisError> {
        let should_construct_value = self.is_constant();

        // adapted for simplicity
        // ...
```

```
        let legendre = self.legendre_qr()?;

        let sqrt_var = Self::new_witness(self.cs(), || {
            self.value()
                .map(|value| value.sqrt().unwrap_or_else(F::zero))
        })?;
        let sqrt_square = sqrt_var.square()?;

        sqrt_square.conditional_enforce_equal(self, &legendre)?;
        sqrt_var.conditional_enforce_equal(
            &Self::zero(), &!legendre.clone()
        )?;

        Ok((legendre, sqrt_var))
    }

    fn legendre_qr(&self) -> Result<Boolean<F>, SynthesisError> {
        self.pow_by_constant(F::MODULUS_MINUS_ONE_DIV_TWO)?.is_one()
    }
}
```

This approach works as follows:

- We first compute the Legendre symbol to determine whether the field element is a quadratic residue (QR).
- If it is a QR, we enforce that the square of the supplied witness equals the original element.
- If not, we enforce that the witness is zero.

Since the Legendre symbol is computed soundly in the circuit, this ensures that our `sqrt` implementation is also sound.

Another smaller but nontrivial challenge lies in implementing the `sgn0` function from Section 4.1 of RFC 9380, which determines the sign of a field element. The difficulty arises because `arkworks` does not expose a method for directly accessing the base field representation (or sign) of a field variable. To overcome this, we define a custom trait `ToBaseFieldVarGadget`, which complements the `FromBaseFieldVarGadget` and enables conversion from any field variable to its base prime field representation:

```
pub trait ToBaseFieldVarGadget<F: PrimeField, CF: PrimeField>: Sized
```

```
{
    type BasePrimeFieldVar: ToBaseFieldVarGadget<F, CF>
                            + FieldVar<F, CF>;

    fn to_base_field_vars(
        &self
    ) -> Result<Vec<Self::BasePrimeFieldVar>, SynthesisError>;
}
```

With these methods and traits, we implement the full map-to-curve algorithm and achieve a one-to-one match with the off-circuit `arkworks` implementation.

5.2.3 Cofactor Clearing

Once we obtain a point on the curve, the final step is to ensure that the point lies in the correct prime-order subgroup. This is accomplished through cofactor clearing. There are two standard approaches for this: (1) multiplying the point by the subgroup cofactor, and (2) applying a curve-specific endomorphism.

In the `arkworks` implementation of off-circuit hash-to-curve for the BLS12 family of curves, the endomorphism-based method is used. To ensure consistency with their implementation, we adopt the same approach in our on-circuit implementation. Since this method is curve-specific - different BLS12 curves use different endomorphisms - we defined a new trait called `CofactorGadget` and implemented it for both BLS12-377 and BLS12-381.

Below is the definition of the trait:

```
pub trait CofactorGadget<
    FP: FieldVar<Self::BaseField, CF>,
    CF: PrimeField
>: CurveGroup
where
    for<'a> &'a FP: FieldOpsBounds<
        'a,
        <Self as CurveGroup>::BaseField, FP
    >,
    <Self as CurveGroup>::Config: SWCurveConfig,
{
    fn clear_cofactor_var(
        point: &ProjectiveVar<Self::Config, FP, CF>,
```

```
    ) -> Result<ProjectiveVar<Self::Config, FP, CF>, SynthesisError>
    {
        let cofactor_bits: Vec<_> =
            <Self::Config as CurveConfig>::COFACTOR
            .iter()
            .flat_map(|value| {
                BigInteger64::from(*value)
                    .to_bits_le()
                    .into_iter()
                    .map(Boolean::constant)
            })
            .collect();

        point.scalar_mul_le_unchecked(cofactor_bits.iter())
    }
}
```

This trait provides a default mechanism for cofactor clearing by performing scalar multiplication using the cofactor bits, which simplifies future implementations for curves that do not support an endomorphism-based method.

With this final component in place, our complete hash-to-curve implementation is finished. It spans roughly 2500 lines of code and is fully reusable for any application that requires hashing to elliptic curves within the R1CS circuit.

5.3 Emulated Field Variable

In this subsection, we first explain how the widely adopted *emulated field* method - originally proposed in xJsnark [KPS18] and used in libraries such as `bellman-bignat` [OWWB20] and `arkworks` - works. We then describe how we encountered, debugged, and ultimately resolved a bug in `arkworks` related to this method.

5.3.1 Mechanisms

Figure 5.1: An emulated field variable consists of n limbs, where each limb initially stores b bits of the target field element.

Figure 5.1 illustrates the structure of an emulated field variable. In this technique, a target field element is represented in a different base field by splitting it into a vector of n limbs $\boldsymbol{l}$. Each limb stores b bits of the original value. The original target field element T can be reconstructed using the following formula:

$$T = \boldsymbol{l}_0 + 2^b \cdot \boldsymbol{l}_1 + \cdots + 2^{(n-1)b} \cdot \boldsymbol{l}_n \mod m$$

where m is the modulus of the target field.

To properly emulate the target field, we want to support common field operations - such as addition, subtraction, multiplication, bitwise operations, and equality checking - within the emulated structure. Below, we focus on addition, multiplication, and equality checking, as these directly relate to the bug encountered in `arkworks`. For more comprehensive coverage, we refer readers to Section IV of [KPS18].

Addition. Addition is relatively straightforward in most cases. Suppose we have two emulated field variables, e_1 and e_2, with corresponding limb vectors $\mathbf{l}$ and $\mathbf{l}'$. The sum of the original target field elements is given by:

$$\begin{aligned} T_1 + T_2 &= \boldsymbol{l}_0 + 2^b \cdot l1 + \cdots + 2^{(n-1)b} \cdot \boldsymbol{l}_n + \boldsymbol{l}'_0 + 2^b \cdot \boldsymbol{l}'_1 + \cdots + 2^{(n-1)b} \cdot \boldsymbol{l}'_n \\ &= (\boldsymbol{l}_0 + \boldsymbol{l}'_0) + 2^b \cdot (\boldsymbol{l}_1 + \boldsymbol{l}'_1) + \cdots + 2^{(n-1)b} \cdot (\boldsymbol{l}_n + \boldsymbol{l}'_n) \end{aligned}$$

From this formula, we observe that each limb may temporarily hold more than b bits (due to the addition). However, as long as each limb remains within the safe bit capacity of the base field - thereby preventing overflows during addition - the above formula correctly computes the sum of the target field elements. If any limb sum exceeds this capacity, the resulting overflow leads to a permanent loss of information about the target field element. This highlights a recurring challenge in emulated field arithmetic: ensuring that limb operations stay within safe bounds to prevent overflows and maintain correctness. To address this, Section IV.A of xJsnark introduces a bit width adjustment operation. This operation resets each limb back to its original bounded form - ensuring that its value remains strictly less than 2^b, as initially intended.

Multiplication. Multiplication differs from addition as it nearly doubles the number of limbs needed to store the result. Consider the following example with two

2-limb numbers:

$$\begin{aligned} a &= \boldsymbol{l}_0 + 2^b \cdot \boldsymbol{l}_1 \\ b &= \boldsymbol{l}'_0 + 2^b \cdot \boldsymbol{l}'_1 \\ a \cdot b &= (\boldsymbol{l}_0 \cdot \boldsymbol{l}'_0) + 2^b \cdot (\boldsymbol{l}_0 \boldsymbol{l}'_1 + \boldsymbol{l}'_0 \boldsymbol{l}_1) + 2^{2b} \cdot (\boldsymbol{l}_1 \boldsymbol{l}'_1) \end{aligned}$$

More generally, for two numbers each with m limbs, their product will have $2m-1$ limbs. A naive approach would compute these limb values using m^2 multiplications, resulting in many unnecessary constraints. Before xJsnark, Karatsuba multiplication was used to reduce this to $m^{\log_2 3}$ multiplications. xJsnark introduces an optimized method that instead supplies the product limbs as witnesses and enforces constraints on them.

To achieve this, we define $\boldsymbol{w}$ as the alleged product limbs and enforce $2m-1$ independent linear equations to verify that:

$$\begin{aligned} \boldsymbol{w}_0 \cdot 1^0 + \boldsymbol{w}_1 \cdot 1^1 + \boldsymbol{w}_2 \cdot 1^2 &= (\boldsymbol{l}_0 \cdot 1^0 + 2^b \cdot \boldsymbol{l}_1 \cdot 1^1) \cdot (\boldsymbol{l}'_0 \cdot 1^0 + 2^b \cdot \boldsymbol{l}'_1 \cdot 1^1) \\ \boldsymbol{w}_0 \cdot 2^0 + \boldsymbol{w}_1 \cdot 2^1 + \boldsymbol{w}_2 \cdot 2^2 &= (\boldsymbol{l}_0 \cdot 2^0 + 2^b \cdot \boldsymbol{l}_1 \cdot 2^1) \cdot (\boldsymbol{l}'_0 \cdot 2^0 + 2^b \cdot \boldsymbol{l}'_1 \cdot 2^1) \\ \boldsymbol{w}_0 \cdot 3^0 + \boldsymbol{w}_1 \cdot 3^1 + \boldsymbol{w}_2 \cdot 3^2 &= (\boldsymbol{l}_0 \cdot 3^0 + 2^b \cdot \boldsymbol{l}_1 \cdot 3^1) \cdot (\boldsymbol{l}'_0 \cdot 3^0 + 2^b \cdot \boldsymbol{l}'_1 \cdot 3^1) \end{aligned}$$

It is evident that the only solution satisfying the above set of equations corresponds to the correct product limbs. More generally, we enforce:

$$\sum_{i=0}^{2m-2} \boldsymbol{w}_i c^i = \left(\sum_{i=0}^{m-1} \boldsymbol{l}_i c^i \right) \cdot \left(\sum_{i=0}^{m-1} \boldsymbol{l}'_i c^i \right), \quad \forall c \in [1, 2m-1].$$

Since addition of variables and multiplication by constants are free in R1CS (i.e., they do not generate new constraints but merely adjust the matrices of R1CS, as described in Definition 7), this technique generates only $O(m)$ constraints.

However, the *multiplication* operation presents a challenge: how can the multiplication result res, which consists of $2m-1$ limbs (denoted as $\boldsymbol{resl}$), interoperate with other emulated field variables that have only m limbs? This necessitates an additional operation called equality checking. Since the result of multiplication spans $2m-1$ limbs, we must reduce it back to m limbs to maintain compatibility. To achieve this, we introduce a witness value r, and enforce the relation $kp + r = res$, where k is an integer witness represented with m limbs, and p is the modulus of the target field (supplied as a public constant with m limbs). The product kp is

computed within the circuit and yields $2m-1$ limbs, while r is a remainder witness with m limbs such that $r < p$. The combined value $kp + r$ thus forms a $2m-1$ limb vector (denoted as $\boldsymbol{kprl}$). The final step is to enforce that $\boldsymbol{resl} = \boldsymbol{kprl}$, which is handled by the equality checking procedure.

Equality Checking. Checking equality between two values is non-trivial since each limb does not necessarily contain exactly b bits - a possibility due to prior operations such as *addition*, provided no overflow occurs. Consequently, a limb-by-limb comparison between res and $kp + r$ is insufficient. However, as observed in xJsnark, the least significant b bits of the first limb of res and $kp + r$ must match, and any excess bits beyond this range can be propagated into the next limb via carries. This motivates a basic equality checking algorithm: at each limb, enforce that the first b bits of $\boldsymbol{kprl}_i + \texttt{carry\_in} + \texttt{pad} - \boldsymbol{resl}_i$ equal zero. Here, `pad` is included to prevent underflow in the subtraction in case $\boldsymbol{resl}_i$ is larger than the sum of the corresponding components in $\boldsymbol{kprl}_i$ and the incoming carry.

However, this naive approach is inefficient. If each limb-wise equality check incurs q constraints, the total cost would be approximately $(2m-1)q$, since there are $2m-1$ limbs. A more efficient strategy again builds upon the observation that in R1CS, addition of variables and multiplication by constants are free. This allows us to *pack* multiple limb equality checks into a single constraint, provided that no overflow occurs during the packing.

For instance, rather than enforcing equality limb-by-limb, we can instead enforce that the lower $2b$ bits of

$$(\boldsymbol{kprl}_i + 2^b \cdot \boldsymbol{kprl}_{i+1}) + \texttt{carry\_in} + \texttt{pad} - (\boldsymbol{resl}_i + 2^b \cdot \boldsymbol{resl}_{i+1})$$

are zero. More generally, we can pack j limbs together and check that the lower jb bits of the accumulated difference are zero, as long as overflow is avoided.

This optimization can be implemented efficiently using a clever trick[1] to achieve the same constraint cost as individual limb checks. Therefore, by packing as many limbs as possible, we can reduce the total number of constraints by a proportional factor.

[1] As demonstrated in `bellman-bignat`: `https://github.com/alex-ozdemir/bellman-bignat/blob/b365e01bfd884ed5bea831f146bc74fe333ec786/src/mp/bignat.rs#L566-L623`. The technique leverages a publicly computable value called `accumulated_extra`, allowing us to avoid expensive bit decomposition (see Section II.A of [KPS18]). Instead of enforcing the lower kb bits to be zero, we simply verify that they match a known, precomputed constant.

5.3.2 Bug and Fix

In this subsection, we describe a bug in the implementation of emulated field variables in `arkworks`. Specifically, we review how `arkworks` implements the emulated field strategy, how the bug was identified, and how it was ultimately fixed.

In `arkworks`, the emulated field variable is implemented similarly to the strategy described earlier. In addition to a vector of limbs, each variable tracks a value called `num_of_additions_over_normal_form`. This value helps determine the maximum possible value that a limb may currently hold. For example, if a limb vector $\boldsymbol{l}$ has a `num_of_additions_over_normal_form` value of k, then each limb can hold a maximum value of up to $(k+1) \cdot 2^b$, where b is the original bit width allocated to each limb.

Another commonly used parameter in the codebase is `surfeit`, which quantifies the excess number of bits a limb might carry beyond the expected width. It is calculated as $\log_2(k+1)$, derived from the maximum value formula above. Since the maximum possible limb value is $(k+1) \cdot 2^b$, taking the logarithm yields $\log_2(k+1)+b$. Here, b is the intended bit width, and $\log_2(k+1)$ represents the bit surplus - i.e., the number of excessive bits.

These values exist to enable fine-grained control over when to perform reduction. As discussed in the context of addition, we must ensure that limb-wise addition does not overflow. This is ensured by checking that the sum of the maximum possible values across limbs stays below the modulus of the base field.

In `arkworks`, the `num_of_additions_over_normal_form` value is updated during addition as follows:

```
// modified for simplicity
pub fn add(&self, other: &Self) -> R1CSResult<Self> {
    // ...

    let mut res = AllocatedEmulatedFpVar {
        cs: self.cs(),
        limbs: self.limbs + other.limbs,
        num_of_additions_over_normal_form: self
            .num_of_additions_over_normal_form
            .add(&other.num_of_additions_over_normal_form)
            .add(&BaseF::one()),
        is_in_the_normal_form: false,
```

```
        target_phantom: PhantomData,
    };

    // ...
}
```

The updated value of `num_of_additions_over_normal_form` is computed as the sum of the `num_of_additions_over_normal_form` values of the two operands, plus 1. This is because the maximum possible value resulting from the addition of two emulated field variables is:

$$(\texttt{self.num\_of\_additions\_over\_normal\_form} + 1) \cdot 2^b + \\ (\texttt{other.num\_of\_additions\_over\_normal\_form} + 1) \cdot 2^b$$

Thus, to maintain correctness, the new `num_of_additions_over_normal_form` is set to the sum of the previous two values, plus one.

With this understanding, we can now discuss the bug. The issue arises in a function called `group_and_check_equality`, which - as the name suggests - performs the grouping-based equality check used in multiplication, as previously described. We discovered that this function was generating unsatisfiable constraints while testing the BLS verification circuit, which uses emulated field variables for multiplication.

Identifying the root of the bug in this function was non-trivial. To aid debugging, we modified parts of the `arkworks-r1cs-std` library (where emulated field variables are implemented) to emit a stack trace when an unsatisfiable constraint occurred. With the help of these traces, we were eventually able to isolate the failure to a specific point where an invalid constraint was generated. It turned out that the problem originated from the following line:

```
// this is the formula we showed before, in which
// left_total_limb and right_total_limb represents
// the grouped limbs.
let eqn_left = left_total_limb + pad_limb +
               &carry_in - right_total_limb;

// this is the clever trick that we mentioned before to avoid
// checking the lower `jb` bits of the `eqn_left` is 0
// - j is `num_limb_in_this_group`
let eqn_right = &carry
```

```
    * BaseF::from(2u64).pow(
        &[(shift_per_limb * num_limb_in_this_group) as u64]
    )
    + remainder_limb;

// this is where the unsatisfiable constraint was enforced
eqn_left.conditional_enforce_equal(
    &eqn_right, &Boolean::<BaseF>::TRUE
)?;
```

After poking around the code and leaving extensive `dbg!` logging, we discovered the root cause of the unsatisfiable constraint: the grouped left limbs (`left_total_limb`), when added with the padding value, overflowed. This overflow caused the computed left-hand side (`eqn_left`) to differ from the right-hand side (`eqn_right`), resulting in an unsatisfiable constraint.

For a while, the reason behind the overflow was unclear - until we noticed that the grouped left limbs were larger than anticipated. It turns out that the number of limbs grouped depends on the following piece of code:

```
// For multiplication
// - bits_per_limb represents 2b
// - shift_per_limb represents b
//
// `arkworks` assumes each limb in the multiplication
// result originally holds 2b bits (see analysis below),
// while `shift_per_limb` is used to compute the shift
// when grouping elements.
let num_limb_in_a_group = (BaseF::MODULUS_BIT_SIZE as usize
    - 1
    - surfeit
    - 1
    - 1
    - 1
    - (bits_per_limb - shift_per_limb))
    / shift_per_limb;
```

This formula attempts to compute how many limbs can safely be grouped for equality checking without causing overflow, considering available bits in the base field and accounting for potential excess bits (`surfeit`).

We empirically found that subtracting 1 from the group variable resolves the bug. However, the underlying question remains: why does this off-by-one error occur during limb calculations? After extensive investigation of the largely undocumented codebase, we identified the root cause based on the following observations:

- **Grouped Limb Size Invariant**. The calculated number of limbs in a group should satisfy an important invariant while trying to group as many limbs as possible: the grouped limb must not exceed the padding value. The padding should be larger than both the left and right grouped limbs to avoid underflow. It is defined as:

```
pad_limb_repr <<= (surfeit
    + (bits_per_limb - shift_per_limb)
    + shift_per_limb * num_limb_in_this_group
    + 1
    + 1) as u32;
```

  Plugging in `num_limb_in_this_group` shows that the padding equals 2^{m-1}, where $m =$ `BaseF::MODULUS_BIT_SIZE`.

- **Maximum of Each Limb in Emulated Multiplication**. For two emulated field variables $\boldsymbol{l}$ and $\boldsymbol{l'}$ each with m limbs and `num_of_additions_over_normal_form` values n and n', the result of their multiplication $\boldsymbol{res}$ is defined as:

$$\begin{cases} \boldsymbol{res}_j = \sum_{i=0}^{j} \boldsymbol{l}_i \cdot \boldsymbol{l'}_{j-i} & \text{for } j \leq m \\ \boldsymbol{res}_j = \sum_{i=j-m}^{m} \boldsymbol{l}_i \cdot \boldsymbol{l'}_{j-i} & \text{for } j > m \end{cases}$$

  Each limb of the result is thus a sum of at most m products. The upper bound of each product is $((n+1)\cdot 2^b)\cdot((n'+1)\cdot 2^b)$. Therefore, each limb in the result can be bounded by:

$$m \cdot ((n+1)\cdot 2^b)\cdot((n'+1)\cdot 2^b) = (m\cdot(n+1)\cdot(n'+1))\cdot 2^{2b}$$

  For simplicity, `arkworks` assumes each limb in the result fits within 2^{2b} (i.e., $2b$ bits). Hence, the result's `num_of_additions_over_normal_form` is set to $m\cdot(n+1)\cdot(n'+1)-1$.

- **Expected Maximum Bit Size of Grouped Limbs**. Based on this, if the `surfeit` value is correct, then a grouped limb (e.g., $\boldsymbol{l}_0 + 2^b \cdot \boldsymbol{l}_1$ for group size

2) should follow the structure illustrated in the following figure:

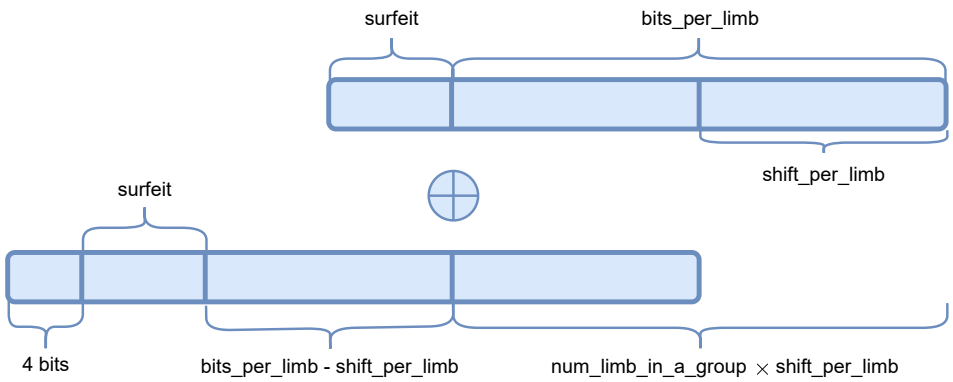


Figure 5.2: A grouped limb with group size = 2, showing bit layout of $l_0 + 2^b \cdot l_1$.

The maximum bit size should then be `BaseF::MODULUS_BIT_SIZE` − 4, with 4 bits reserved to avoid overflow when adding padding, carry-in, and grouped limbs. If `surfeit` is underestimated, grouped limbs may occupy more than the allowed space, leading to overflows.

Based on the above reasoning, we hypothesize that the root cause is an underestimated `surfeit`, which leads to an overestimated `num_limb_in_a_group`. Our suspicion deepened upon examining the following code involved in computing multiplication results:

```
// modified for clarity
Ok(AllocatedMulResultVar {
    //...
    limbs: prod_limbs,
    // num_of_additions_over_normal_form
    prod_of_num_of_additions: (self.num_of_additions_over_normal_form
        + BaseF::one())
        * (other.num_of_additions_over_normal_form + BaseF::one()),
    //...
})
```

This piece of code is suspicious because:

- After constructing the `AllocatedMulResultVar`, the `group_and_check_equality` function is typically called immediately using the `limbs` and

`num_of_additions_over_normal_form` (named `prod_of_num_of_additions` in the code) from this struct. This suggests that the fields of `AllocatedMulResultVar` are likely the direct inputs to the function when it was generating unsatisfiable constraints.

- The `num_of_additions_over_normal_form` in `AllocatedMulResultVar` was underestimated - the critical multiplication by the number of limbs was missing.

Ultimately, we fixed the code as follows:

```
// modified for clarity
Ok(AllocatedMulResultVar {
    //...
    limbs: prod_limbs,
    // we slightly overestimate it but it does no harm
    prod_of_num_of_additions: (self.num_of_additions_over_normal_form
        + BaseF::one())
        * (other.num_of_additions_over_normal_form + BaseF::one())
        // newly added
        * BaseF::from((params.num_limbs) as u32),
    //...
})
```

After applying this fix (along with additional corrections related to subtraction, which suffered from the same root cause), we verified that the `group_and_check_equality` function no longer produces unsatisfiable constraints. Finally, we upstreamed the patch, including all fixes, to `arkworks`[2].

5.4 Optimizations

In this section, we present two optimizations we applied to the circuit to reduce the number of constraints: the first optimizes pairing, and the second optimizes the folding circuit.

[2]The pull request is available at `https://github.com/arkworks-rs/r1cs-std/pull/157`.

5.4.1 Pairing

Recall the definition of the `Verify` algorithm for BLS signatures, which involves checking the equality of a bilinear pairing:

$$\text{Verify}(\sigma, m): \quad e(pk, H(m)) = e(g, \sigma)$$

This can be naively implemented in `arkworks`'s R1CS as follows:

```
let signature_paired = bls12::PairingVar::pairing(
    G1PreparedVar::<SigCurveConfig, FV,
↪   CF>::from_group_var(&parameters.g1_generator)?,
    G2PreparedVar::<SigCurveConfig, FV,
↪   CF>::from_group_var(&signature.signature)?,
)?;
let aggregated_pk_paired = bls12::PairingVar::pairing(
    G1PreparedVar::<SigCurveConfig, FV,
↪   CF>::from_group_var(&pk.pub_key)?,
    G2PreparedVar::<SigCurveConfig, FV,
↪   CF>::from_group_var(&hash_to_curve)?,
)?;

signature_paired
    .is_eq(&aggregated_pk_paired)?
    .enforce_equal(&Boolean::TRUE)?;
```

However, this implementation can be further optimized. The pairing operation consists of two stages: the Miller loop (which evaluates a function f on the given G_1 and G_2 points) and the final exponentiation, which raises the result to the power $\frac{(p^k-1)}{r}$. For more details, we refer the reader to Section 3 of [Hou23].

A useful observation is that if we denote the Miller loop as $M(\cdot, \cdot)$, the verification equation can be rewritten as:

$$\text{Verify}(\sigma, m): \quad M(pk, H(m))^p = M(g, \sigma)^p$$

where p is the exponent used in the final exponentiation and is the same on both

sides. Since the exponent is the same, the equation can be simplified as:

$$\begin{aligned}
\text{Verify}(\sigma, m): \quad & M(pk, H(m))^p = M(g, \sigma)^p \\
\iff \quad & (M(pk, H(m)) \cdot M(g, \sigma)^{-1})^p = 1 \\
\iff \quad & (M(pk, H(m)) \cdot M(-g, \sigma))^p = 1 \qquad \text{(by bilinearity of pairing)}
\end{aligned}$$

This means we only need to perform a single final exponentiation on the product of two Miller loop evaluations, instead of computing two separate final exponentiations and comparing the results.

A preliminary study evaluating the $\mathcal{BLS}$ circuit over a native field shows that this optimization reduces the number of constraints by approximately 20%.

5.4.2 Folding

In `sonobe`, IVC is modelled through the following trait:

```
/// slightly modified for clarity
///
/// FCircuit defines the trait of the circuit of the F function,
/// which is the one being folded.
/// The parameter z_i denotes the current state, and z_{i+1} denotes
/// the next state after applying the step.
pub trait FCircuit<F: PrimeField>: Clone + Debug {
    type Params: Debug;
    type ExternalInputs;
    type ExternalInputsVar;

    /// returns a new FCircuit instance
    fn new(params: Self::Params) -> Result<Self, Error>;

    /// returns the number of elements in the state of the FCircuit,
    /// which corresponds to the FCircuit inputs.
    fn state_len(&self) -> usize;

    /// generates the constraints for the step of F for the given z_i
    /// and returns z_{i+1}
    fn generate_step_constraints(
        &self,
```

```
        cs: ConstraintSystemRef<F>,
        i: usize,
        z_i: Vec<FpVar<F>>,
        external_inputs: Self::ExternalInputsVar,
    ) -> Result<Vec<FpVar<F>>, SynthesisError>;
}
```

To use `sonobe`, we must define a circuit that implements this trait. The critical design task is to determine what states must be returned (i.e., the return value of `generate_step_constraints`) and what constraints must be enforced to ensure the returned state is valid.

Compared to the abstraction of the folding SNARK introduced in Figure 4.3, a key distinction in `sonobe`'s interface is that `generate_step_constraints` requires the next state to be returned directly. This implies that the next state (i.e., the checkpoint, in our context) must be passed as part of the external inputs. One thing not obvious from the above API is that these external inputs - represented by `ExternalInputsVar` - are witnesses, just like the signature in our $\mathcal{FOLD}$ circuit.

This design choice significantly influences how we implement `ExternalInputsVar`. Since it's a witness, the circuit must enforce constraints to validate its properties. Specifically, the following conditions must be verified:

- The checkpoint's signature must lie on the elliptic curve and belong to the correct subgroup.
- The next committee's public keys must also lie on the elliptic curve and belong to the correct subgroup.

Unfortunately, verifying these conditions - especially when using emulated field variables - is costly. Our preliminary measurements indicate that ensuring a single point lies in the correct subgroup and on the curve costs approximately 25 million constraints. Worse yet, the number of public keys scales with the committee size. For example, in Ethereum, the light client protocol has a committee size of 512, so verifying all public keys would incur an overwhelming number of constraints.

However, this cost can be avoided. Recall the security definition of the light client protocol in Definition 1. The protocol guarantees that each checkpoint is valid, and specifically that the public keys of the next committee in a checkpoint are correct because each checkpoint (except the genesis checkpoint, which is automatically trusted) is signed by the supermajority of the current committee. This signature is checked within the circuit and covers the entire checkpoint (including the public

keys of the next committee). Therefore, the soundness of the light client protocol already implies that the public keys are valid.

As a result, we only need to enforce that the signature lies on the elliptic curve and in the correct subgroup - this check does not scale with committee size and keeps constraint costs manageable even when supporting large committees.

Chapter 6

Evaluation

In this chapter, we evaluate the performance of our light client protocols (with and without LMF) and compare the Levelled Merkle Forest (LMF) with the Merkle tree. All experiments were conducted on an AWS r8g.24xlarge instance with 768 GB of RAM, 800 GB swaps, and 96 CPU cores.

6.1 Light Client Protocol

In this section, we break down the number of constraints generated by each component and evaluate the runtime of folding steps, SNARK proof generation, proof verification, and peak memory usage during proving. Due to time and memory constraints, we extrapolate key performance metrics, such as final proof generation time and peak memory usage. Specifically, for extrapolation, we run SNARK proofs on a dummy circuit with the same state length, varying the number of constraints generated. This extrapolation remains accurate because SNARK proving time and memory usage primarily depend on the number of constraints and variables, independent of the specific computation being proved.

Experiment Setup. The folding-based SNARK was run on MNT4-753 and MNT6-753 with Nova [KST22] as the IVC scheme and Groth16 [Gro16] as the SNARK scheme. We analyze both the light client protocol with and without LMF. For the light client protocol with LMF, we use a checkpoint chain size of 1024, which is sufficient to cover approximately 3 years of checkpoints for real-world blockchains such as Ethereum and Sui. For LMF, we use $q = 7$ (one of the optimal parameter choices) and compute k accordingly.

Experiment Procedure. We evaluate three committee sizes: $n = 128$, $n = 256$,

and $n = 512$, which are representative of committee sizes in Ethereum (512 members) and Sui (roughly 128 members). For each committee size, we measure the time to generate five folding steps. After folding, we measure the time to produce the final SNARK proof. Additionally, we record the verification time and peak memory usage throughout the entire process.

6.1.1 Results

Number of Constraints

Component	Constraints	Percentage (%)
Witness Check	24827996	20.00
Public Key Aggregation	8419348	6.78
Hash to Curve	27986621	*22.54*
Pairing	57011277	**45.92**
Others	5874757	4.73
Total	124142647	100

Table 6.1: Constraint contributions by component for the light client protocol (committee size $n = 512$)

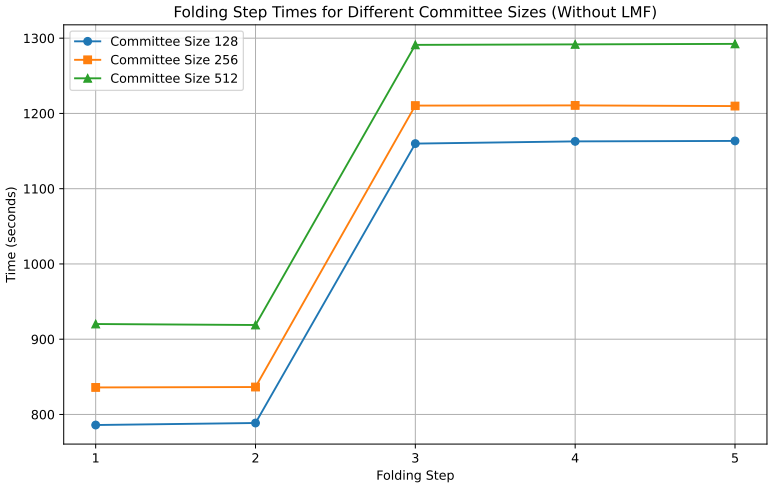
Component	Constraints	Percentage (%)
Witness Check	24827996	19.66
Public Key Aggregation	8419348	6.67
Hash to Curve	27986621	*22.16*
Pairing	57011277	**45.14**
Levelled Merkle Forest	295659	0.23
Others	7750341	6.14
Total	126291242	100

Table 6.2: Constraint contributions by component for the light client protocol with LMF (committee size $n = 512$)

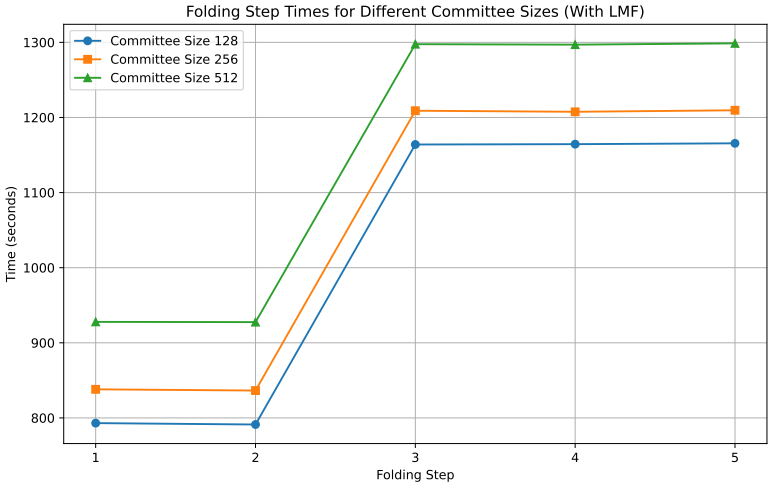
Table 6.1 and Table 6.2 show the breakdown of the number of constraints contributed by each major component of the SNARK circuit. Across both variants of the protocol, **pairing operations are the dominant cost**, accounting for approximately 46-47% of the total constraints. A closer inspection reveals that this cost is roughly evenly split between the Miller loop and the final exponentiation, the two core subroutines of pairing. This highlights a crucial performance bottleneck and indicates that more optimized R1CS-level pairing gadgets could yield significant gains. The second most expensive operation is *hash-to-curve*, accounting for over 22% of the total. A closer look reveals that the overhead largely stems from evaluating the Blake2s hash function within the circuit, particularly during the expansion step in the hash-to-field. Switching to a SNARK-friendly hash such as Poseidon

[GKR+21] could drastically reduce this portion of the constraint budget. The witness check comes in third, consuming around 20%. This is largely unavoidable, as it ensures the correctness of witnesses. Interestingly, when we integrate the Levelled Merkle Forest (LMF), its contribution to the overall constraint count is negligible - only 0.23% - yet it enables us to offload expensive SNARK proof generation during proving. This efficiency strongly justifies its inclusion in the design, especially when balanced against the minimal overhead introduced.

Folding



(a) The folding time of the light client protocol.



(b) The folding time of the light client protocol with LMF.

Figure 6.1: The folding time of the light client protocol with and without LMF.

Folding Time. Figure 6.1 presents the folding step times for the light client proto-

col, both with and without LMF, across committee sizes of 128, 256, and 512.

Across different protocols, the LMF introduces negligible overhead to the folding step times. For example, at committee size 128, the times without LMF are 785.966, 788.646, 1159.922, 1162.862, and 1163.467 seconds, compared to 793.072, 791.220, 1164.016, 1164.472, and 1165.589 seconds with LMF. This corresponds to overheads of 0.90%, 0.33%, 0.35%, 0.14%, and 0.18%, respectively. For committee sizes 256 and 512, the maximum observed overhead is less than 1%, confirming that LMF has minimal impact on computational efficiency.

Across different committee sizes, folding time scales moderately with committee size while maintaining overall efficiency. For instance, in step 3 without LMF, the folding time increases from 1159.922 seconds at committee size 128 to 1291.101 seconds at size 512—a rise of 11.31%. Across all steps, the maximum increase from size 128 to 512 is 17.07%, demonstrating that the protocol scales well and incurs low computational overhead even for larger committee sizes. This moderate increase is especially acceptable when contrasted with the significantly higher SNARK proving cost discussed in Figure 6.2.

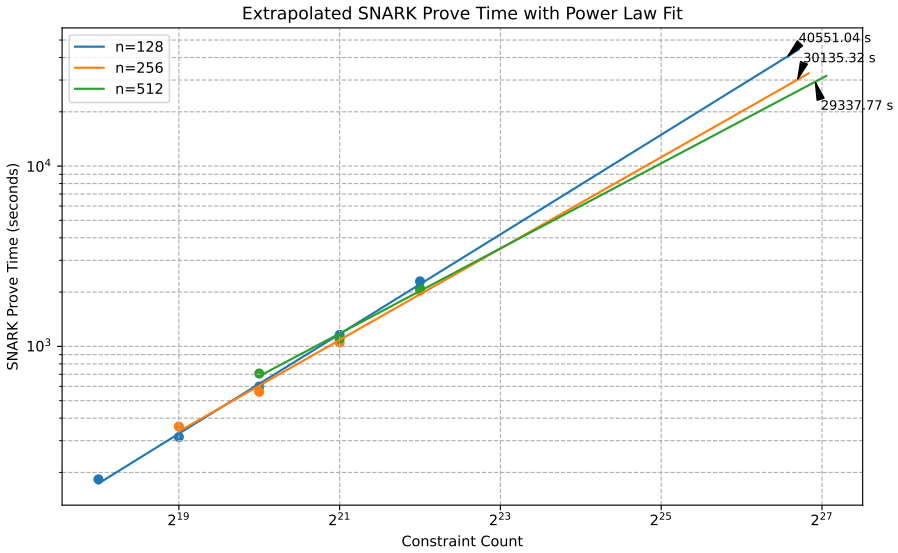
Fast Proving in Steps 1-2. A notable pattern is the sharp increase in folding time at the third step across all committee sizes. While steps 1 and 2 consistently take under 1000 seconds, step 3 jumps above 1000 seconds - regardless of whether LMF is used - and subsequent steps remain steady at this elevated level. One plausible explanation is that the first folding step, being the base case, involves fewer checks and therefore completes faster. However, the relatively short time for the second step remains unexplained and warrants further investigation. Future work should include extensive profiling of the folding workflow to pinpoint the cause of this behaviour.

SNARK

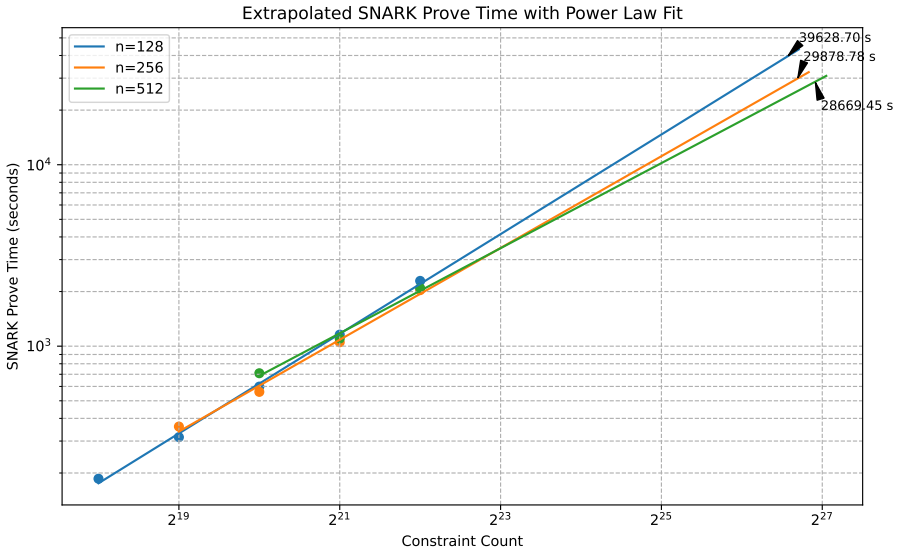
Proving Time. Figure 6.2a and Figure 6.2b illustrate the SNARK proving time required to generate the final proof. Overall, both plots demonstrate sub-linear growth in proving time, consistent with the results reported in Nova's experiments.

Across different protocols, the light client protocols with and without LMF exhibit similar trends in proving time. The minor variations observed can be attributed to extrapolation and timing fluctuations. Given the scale of the measurements, these differences are minimal, within 1%.

Across different committee sizes, an interesting trend emerges. Initially, the protocol with the largest committee size ($n = 512$) incurs the highest proving time for both



(a) Extrapolated SNARK proving time of the light client protocol.



(b) Extrapolated SNARK proving time of the light client protocol with LMF.

Figure 6.2: Extrapolated SNARK proving time of the light client protocol with and without LMF.

the LMF and non-LMF variants. However, as the number of constraints increases, its proving time grows more slowly than that of smaller committee sizes ($n = 128$ and $n = 256$). As a result, it eventually becomes the most efficient, completing the proving process faster than the others at larger constraint scales. This suggests that, under heavier computational loads, the proving time scales more favourably with larger committee sizes.

The underlying reason for this counterintuitive behaviour remains an open question.

It could stem from characteristics of the proof system itself or specifics of the implementation[1]. A closer look at the time breakdown reveals that the spike in proving time for $n = 128$ at a constraint count of 2^{22} coincides with increased setup time, particularly for SNARK parameter initialization. This trend also echoes the peak memory usage pattern discussed later in Figure 6.4.

Compared to the folding time, the proving time is significantly longer - approximately 30 times greater, based on extrapolated measurements. This further supports the use of LMF, as it alleviates the need for prover nodes to perform SNARK compression at every checkpoint, thereby improving the protocol's practicality. For example, assuming a checkpoint generation rate of one checkpoint per day (as in Sui), a chain of 365 checkpoints representing 1 year of data could be compressed and proved in under one week when LMF is used.

Verification Time. Figure 6.3a shows the averaged proof verification time of our light client protocol. Compared to the naive approach described in Protocol 1, our protocol maintains a constant proof size regardless of the number of checkpoints. The verification time scales roughly linearly with the committee size, reaching 3.12 seconds for a committee of size 512. Overall, the 1-3 second latency is practical and responsive for common light client use cases, such as verifying transaction status.

Figure 6.3b presents the averaged verification time for our light client protocol with LMF integration. As observed previously, the verification time scales linearly with committee size, reaching 3.21 seconds at size 512. Despite a minor increase of 0.1 seconds in verification time, the LMF offers substantial benefits - such as reducing proving time and enabling fast verification once the LMF state is verified - which fully justifies this slight overhead.

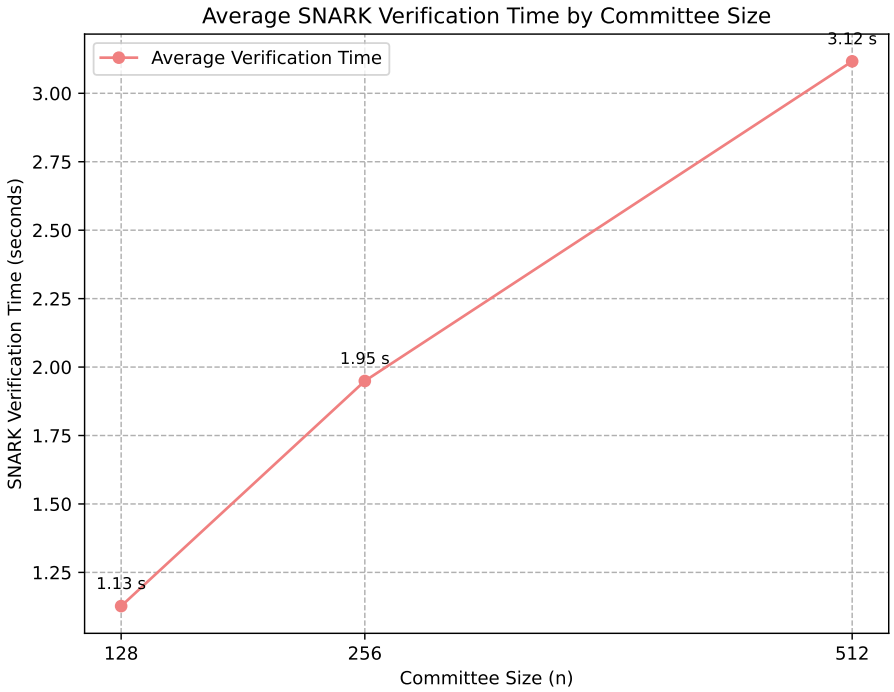
Peak Memory Usage

Figure 6.4a and Figure 6.4b illustrates the peak memory usage of the light client protocol across three different committee sizes.

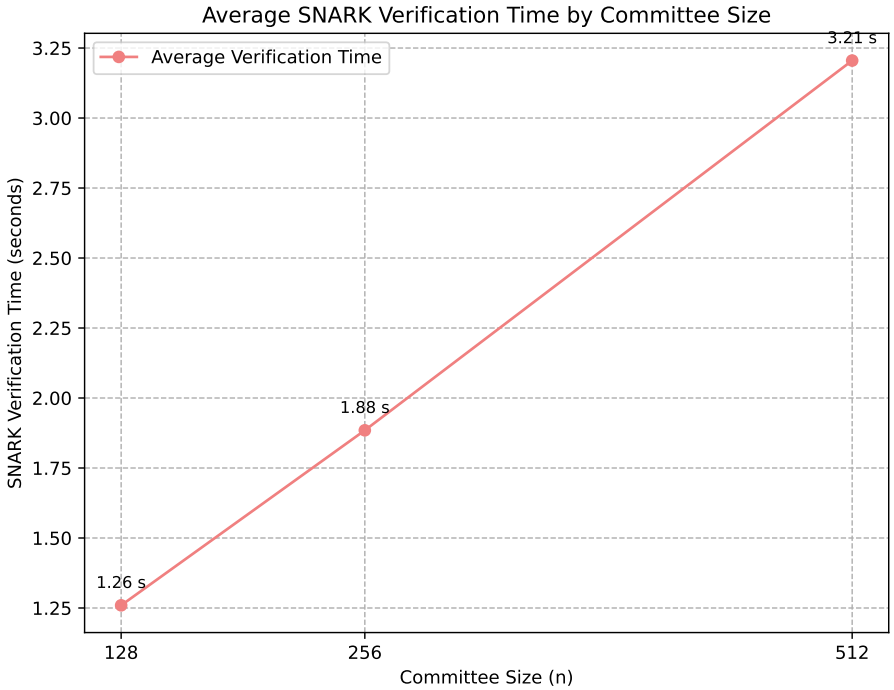
Across different protocols, the difference in peak memory usage between versions with and without LMF remains minimal - within approximately 1% across all committee sizes - demonstrating once again the low overhead and practicality of integrating LMF into the protocol.

Across different committee sizes, the memory usage trends closely mirror those ob-

[1]For instance, developers of the `sonobe` are investigating similar regressions in performance of their on-chain decider: `https://github.com/privacy-scaling-explorations/sonobe/issues/211`.



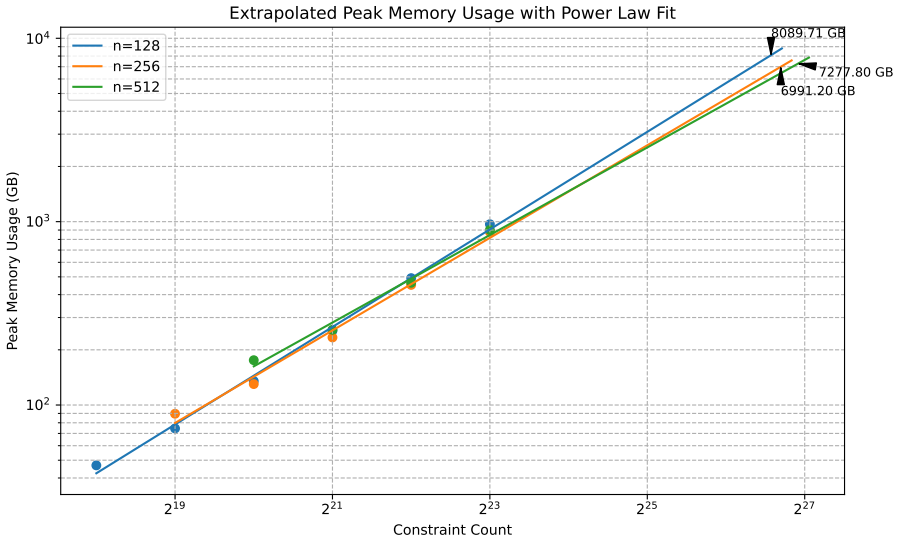
(a) The proof verification time of the light client protocol, averaged across 5 runs for each committee size.



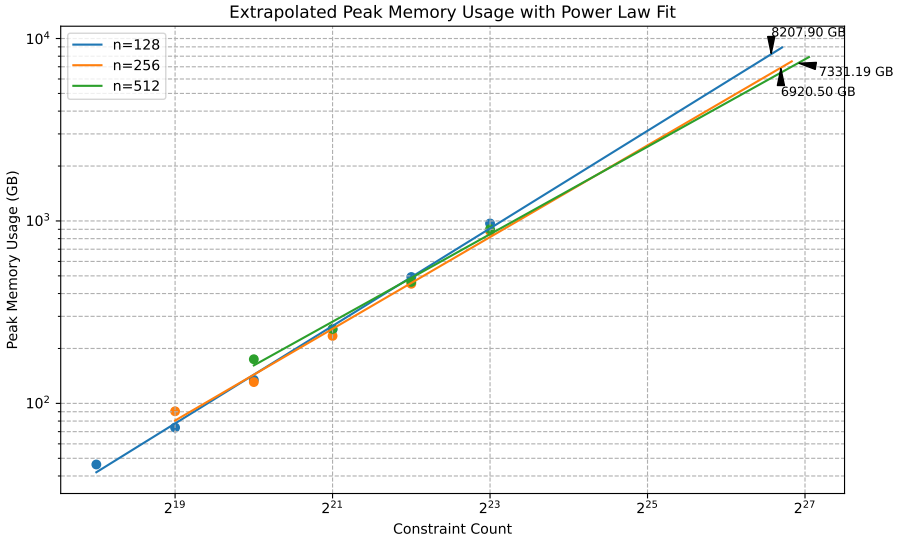
(b) The proof verification time of the light client protocol with LMF, averaged across 5 runs for each committee size.

Figure 6.3: The proof verification time of the light client protocol with and without LMF.

served in Figure 6.2. Initially, the protocol with $n = 128$ consumes less memory than configurations with larger committees. However, starting at a constraint count



(a) Extrapolated proving memory usage of the light client protocol.



(b) Extrapolated proving memory usage of the light client protocol with LMF.

Figure 6.4: Extrapolated proving memory usage of the light client protocol with and without the LMF structure. Due to the inherent overhead of basic witness checks generated by each circuit, some x-axis values are unavailable for certain circuits. For example, the green line begins at $x = 2^{20}$, whereas the other line starts earlier.

of 2^{22}, its memory usage surpasses that of $n = 256$ and $n = 512$. This high growth rate leads to $n = 128$ ultimately requiring more memory than $n = 256$ and $n = 512$ at extrapolated scales. This indicates that, under the same constraint load, $n = 128$ is more memory-intensive.

Understanding this unexpected behaviour remains an open question. Given that

similar trends appear in the proving time analysis, we hypothesize that the SNARK parameter generation and proving stages for $n = 128$ may exhibit characteristics that result in a disproportionately high memory growth rate. Future work should investigate the internals of these stages to better understand the cause.

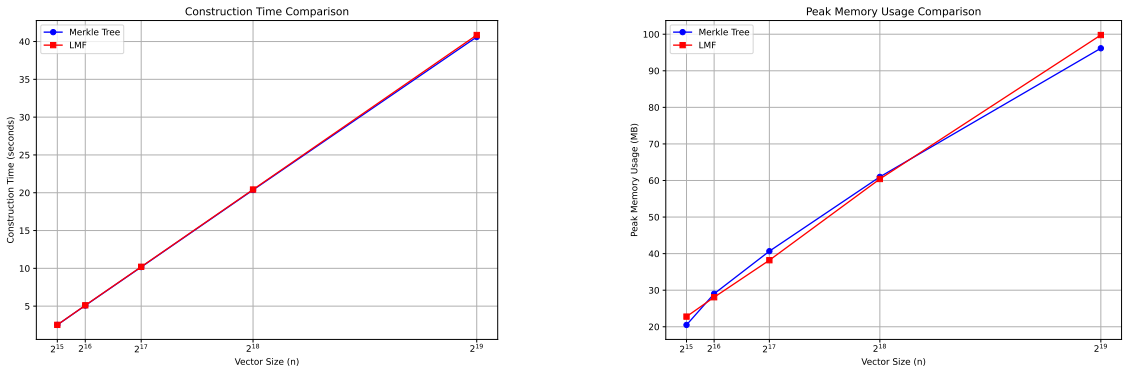
6.2 Levelled Merkle Forest

In this section, we evaluate the construction time, proof size, proof time, and peak memory usage of the Levelled Merkle Forest (LMF), and show that our implementation achieves performance comparable to traditional Merkle trees, while also enabling reduced proof sizes through variable-length proving.

Experiment Setup. We use the Poseidon hash function, instantiated over the scalar field of the BLS12-381 curve. For LMF, we use $q = 7$ (one of the optimal parameter choices) and compute k based on the size of the committed vector.

Experiment Procedure. We vary the vector size from $n = 2^{15}$ to $n = 2^{19}$ and measure the performance impact. For each n, we construct both a standard Merkle tree and an LMF, select 10 random leaves, and measure the average time to generate fixed-size proofs for these leaves. Additionally, we provide an analytical study of how the size of a variable-length proof changes as a function of the index being proved.

6.2.1 Construction



(a) The construction time of Merkle tree and LMF.

(b) The construction memory usage of Merkle tree and LMF.

Figure 6.5: Metrics about the construction of Merkle tree and LMF.

Construction Time: Both Merkle Tree and LMF construction times scale approximately linearly with the vector size n (e.g., from 2.51s to 5.08s for the Merkle tree,

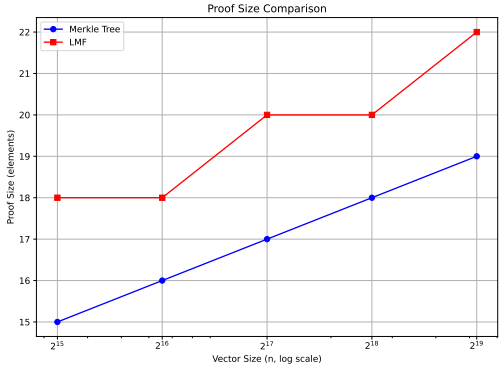
and 2.52s to 5.11s for the LMF when n increases from 2^{15} to 2^{16}), which aligns with our asymptotic analysis.

LMF construction is slightly slower than that of a traditional Merkle tree (e.g., 40.86s vs. 40.61s for $n = 2^{19}$). This minor overhead is likely due to the additional bookkeeping required in LMF's levelled structure. This effect is also reflected in the higher peak memory usage observed in Figure 6.5b. Nevertheless, the construction time difference is small - typically within 1-2% - demonstrating that LMF provides construction performance comparable to that of traditional Merkle trees.

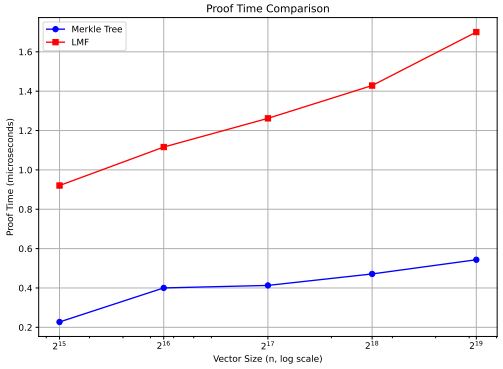
Peak Memory Usage. Similar to construction time, the peak memory usage of both the Merkle tree and LMF grows roughly linearly with n. Interestingly, LMF initially consumes more memory, likely due to the computed value of k being based on upper bounds for the number of states in the R1CS. These conservative estimates may lead to allocating more trees than necessary, inflating memory usage.

In the mid-range of n, LMF achieves on-par or slightly better memory performance compared to the Merkle tree. However, for large values (e.g., $n = 2^{19}$), LMF's memory usage becomes noticeably higher. This overhead again stems from its levelled construction: tree digests must be cached in a vector to build the next level, increasing both memory usage and, slightly, construction time. Despite this, the overall memory usage difference remains modest, within 5%, indicating that LMF maintains competitive memory efficiency while offering structural advantages for proof size and verification.

6.2.2 Fixed-Size Proof



(a) The fixed-size proof size of Merkle tree and LMF.



(b) The fixed-size average proof time of Merkle tree and LMF.

Proof Size: LMF fixed proof sizes are generally slightly larger than those of Merkle trees (e.g., 22 vs. 19 for $n = 2^{19}$). However, they do not consistently grow with n

- for instance, the size remains constant at 18 for $n = 2^{15}$ and 2^{16}, and at 20 for $n = 2^{17}$ and 2^{18}. Overall, the results show that LMF achieves proof sizes comparable to traditional Merkle trees.

Proof Time: Proof generation times for both structures are in the microsecond range, but LMF proof times are slightly higher. These higher proof times may be attributed to the computational overhead of handling its levelled structure.

6.2.3 Variable-Length Proof

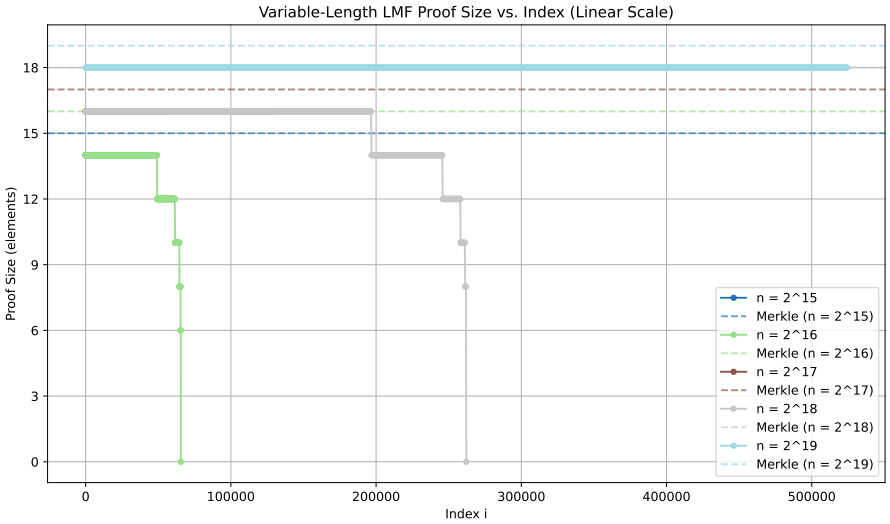
Recall from Section 4.3.4 that for any $\mathbf{v}_i$ such that $n-i \leq \left(\frac{q+1}{2}\right)^l$, the hash of its M_{l-1} is stored in M_l within the state. This implies that a proof of length approximately $\log_2 \frac{q+1}{2} \cdot l$ suffices for $\mathbf{v}_i$. Figure 6.7 illustrates how the variable-length proof size changes with the index being proven, for vectors of length 2^{15} to 2^{19}.

Step-like Behavior. The step-like behaviour observed in the graph arises from discrete transitions in the level l, determined by the condition $n - i \leq \left(\frac{q+1}{2}\right)^l$. As i increases from 0 to $n-1$, $n-i$ decreases, crossing powers of $\left(\frac{q+1}{2}\right)$ and decrementing l. Each decrement in l decreases the proof size, resulting in the characteristic jumps from $2l + 2$ to $2l$.

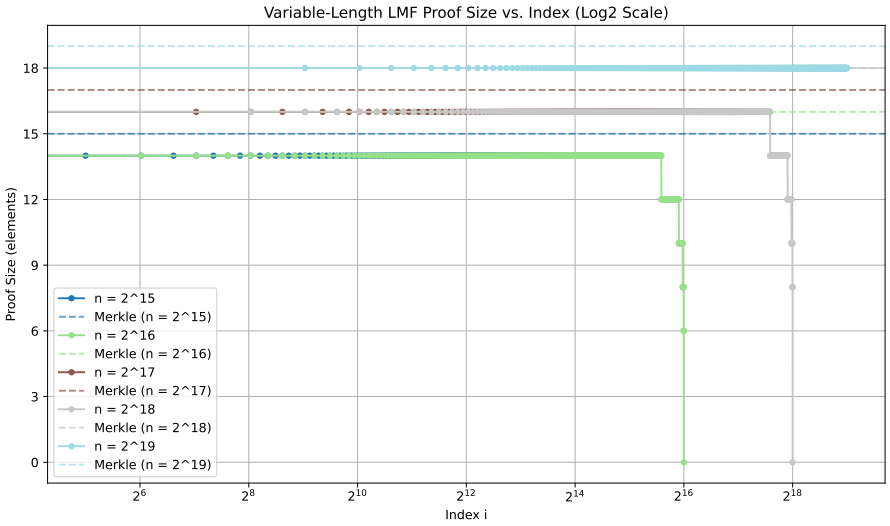
Proof Size. A key trend is that variable-length LMF proofs are consistently smaller than those of Merkle trees across all indices. For example, when $n = 2^{18}$, the proof size starts at 16 for $i = 0$, compared to 18 for Merkle trees. As i approaches n, the proof size rapidly shrinks due to the logarithmic nature of level l. In such regions, LMF reaches peak efficiency, producing proofs of size 6 or 8 - significantly smaller than the $\log_2 n$ required by Merkle trees - and even as small as 0 for indices extremely close to n (within $(q + 1)/2$).

When n is not a power of $\frac{q+1}{2}$, variable-length proofs offer limited improvements. For instance, with $n = 2^{19}$, the proof size remains constant across indices - even though still smaller than Merkle tree proofs - because n is rounded up to the nearest power of $\frac{q+1}{2}$ for correctness when producing the proof. As a result, even the largest index $2^{19} - 1$ maps to level $l = 9$, corresponding to a proof size of 18. Nevertheless, this worst-case scenario still outperforms Merkle trees.

This efficiency underscores the practical advantage of LMF, especially in light client protocols where recent states (i.e., indices near n) are queried more frequently.



(a) Size of the variable-length proof vs. the index being proven.



(b) Size of the variable-length proof vs. the index being proven in log2 scale.

Figure 6.7: Size of the variable-length proof vs. the index being proven. For each n, the plot is generated by sampling 1000 evenly spaced indices from 0 to $n-1$. When n is not a power of $\frac{q+1}{2}$, it is rounded up to the next power of $\frac{q+1}{2}$ when determining the size of the variable length proof, mimicking the behaviour of the actual LMF. The dashed line indicates the size of a traditional Merkle proof, while the solid line represents the size of the variable-length proof.

Chapter 7

Discussion

In this chapter, we discuss several open problems and directions for improvement in our design.

7.1 Forking

We proved the security of our light client protocol using folding-based SNARKs in Section 4.2.3. This proof relies on the underlying security assumptions of the light client protocol as defined in Definition 1. However, this security can be violated in the presence of a fork.

Formally, a fork occurs when a supermajority of the committee at checkpoint cp_i signs two conflicting checkpoints cp_{i+1} and cp'_{i+1}. This event causes the chain to split into two branches, $\mathcal{C}$ and $\mathcal{C}'$, where one continues from cp_{i+1} and the other from cp'_{i+1}. Forks may arise from both technical and non-technical causes, such as the Ethereum DAO incident [SMG+17], where the Ethereum community voted to revert the state to before an exploit that resulted in a 50M USD theft.

In such scenarios, our light client protocol would accept proofs from both chains, allowing a light client on chain $\mathcal{C}$ to accept a proof for a checkpoint $cp' \in \mathcal{C}'$ and mistakenly assume cp' is valid on $\mathcal{C}$.

To mitigate the impact of forks, both chains can continue using our light client protocol by redoing the setup phase. Instead of continuing with the original $gen \leftarrow$ `FoldingSNARK.Setup` as the genesis state, each chain can rerun the setup and use the first divergent checkpoint as the new genesis. This ensures that a light client only accepts proofs generated from the correct fork. However, a cautious light client may additionally verify the validity of the new genesis state by checking it against

the old light client protocol on the non-forked chain.

7.2 Practicality

In our evaluation, we demonstrate the performance of the proposed LMF data structure and assess the efficiency of the light client protocol. One major limitation, however, lies in the high SNARK proving time and substantial memory requirements during proving. While the proving time remains within a manageable range - typically under half a day - and aligns well with the checkpoint frequency of systems like Ethereum and Sui (approximately once per day), the memory demand presents a significant practical challenge. Most commercially available machines are unable to execute the folding-based SNARK. The closest AWS instances capable of running our circuit - `u-9tb1.112xlarge` and `u7i-8tb.112xlarge` - cost around $80 per hour, rendering them financially impractical for many real-world deployments.

Despite these constraints, the proposed scheme represents a valuable first-of-its-kind baseline, exposing the limitations of current techniques and outlining directions for future improvement. Importantly, while the proving process is expensive, it is a one-time cost: a single proof can be reused by an unlimited number of light clients, which may justify the initial investment. However, the question of how to incentivize participants to serve as prover nodes under such high costs remains open. Addressing performance bottlenecks is another promising avenue, as efficiency improvements could substantially lower operational costs and enhance the system's practicality.

7.3 Future Work

Parallelism. The performance of the light client protocol in Protocol 4 (folding-based SNARK + LMF) can potentially be improved through parallelization. Although the IVC process is inherently sequential, it is possible to divide the checkpoint chain into multiple segments and generate proofs for each segment using different checkpoints as genesis. Once each segment is proven, a final SNARK proof can be generated to aggregate the resulting LMF digests into another Merkle tree. For verification, the prover provides the light client with this final proof, the Merkle root, the LMF state relevant to the client's request, and an LMF proof for the queried checkpoint. Future work can explore the performance trade-offs of this approach for both provers and light clients.

Folding at Different Levels. Our solution currently relies on folding-based IVC techniques to aggregate multiple instances. However, there are other layers where

aggregation could occur. For instance, BLS signatures support aggregation, enabling the verification of multiple signatures simultaneously. This raises the question: should we first aggregate using BLS signature techniques and then generate the proof, or should we perform all aggregation through folding-based SNARKs? Based on current observation, given the already high memory usage, employing BLS aggregation within the circuit may not be feasible, as it increases the number of constraints.

Co-design of Light Client Protocol and SNARK Proof System. Our current implementation assumes that existing signature and hash schemes in the light client protocol cannot be changed. As a result, we rely on field emulation for general compatibility and use Blake2s as the underlying hash function. However, co-designing the light client protocol in tandem with the SNARK system could yield substantial efficiency gains. For instance, adopting transitive signature schemes - such as those utilized in [AKMW24], which modify BLS signatures to sign the quotient of two hashes instead of a single one - could enable efficient aggregation of signatures. Alternatively, signature schemes like Schnorr [Sch91], which avoid expensive pairing operations, could eliminate the need for field emulation. Moreover, embedding a Merkle accumulator or the LMF state directly within the checkpoint data structure would allow Merkle or LMF state verification to occur outside the R1CS circuit, reducing constraint counts. Finally, using SNARK-friendly hash functions such as Poseidon [GKR+21] could substantially reduce the number of constraints. However, further research is needed to explore how Poseidon can be integrated into hash-to-curve, as current standards do not define support for hash functions that operate over finite fields. Future work can explore such co-design opportunities to create light client protocols that are natively optimized for SNARK-based proving systems.

Better Proof Systems. Our implementation uses the classic Groth16 [Gro16] proof system, primarily because it is the only system that properly implements the `SNARK` trait in `arkworks`, which is required by `sonobe` to interface with the proof system. This choice limits our ability to select efficient cycles of elliptic curves since we must find two curves that satisfy both the cycle condition and have pairing support. Consequently, we use the computationally expensive MNT4_753 and MNT6_753 curve pair [BSCTV14]. Future work can enhance the `arkworks` ecosystem by implementing the `SNARK` trait for proof systems like Spartan[1], unlocking performance gains by allowing the use of more efficient cycles like the Pasta curves [Hop20], which offer smaller field sizes without sacrificing security. This leads to faster proving time, smaller proof sizes, and more efficient verification.

[1] `https://github.com/arkworks-rs/spartan`

Improved Pairing Algorithms for Emulated Field. When implementing R1CS, developers often use tricks to optimize circuits by minimizing the number of multiplication gates, under the assumption that additions and constant multiplications are free. This assumption holds for native field elements. However, when using field emulation (as in `arkworks`, following [KPS18]), this no longer applies - constant operations are just as expensive as variable ones. Interestingly, `arkworks` offers a special operation `mul_without_reduce`, which returns an unreduced variable, after multiplication, supporting further additions. Thus, a pattern like `mul_without_reduce` + `additions` + `reduce` could be more efficient than `mul` + `additions`[2]. Future work can exploit these characteristics to design more efficient pairing algorithms tailored for emulated fields.

Empirical Study on Access Patterns. In Section 4.3.1, based on observations in [AKMW24], we hypothesize that access to checkpoints is non-uniform - recent checkpoints are accessed more frequently than older ones. However, there is currently no empirical study that formally confirms or models this behaviour. Conducting such a study would provide valuable data to model access distributions and inform the design of variable-length proofs.

[2]Currently, `arkworks` does not implement addition on unreduced results correctly. We highlighted this in `https://github.com/arkworks-rs/r1cs-std/pull/157`.

Chapter 8

Conclusion

In this work, we propose a protocol designed to enable light clients to efficiently verify blockchain state, with a particular focus on committee rotation. By utilizing folding-based SNARK and a novel LMF data structure, we address the challenges of verifying committee rotations in resource-constrained environments. Our approach not only minimizes verification time but also allows variable-length proofs, optimizing the efficiency of light-client protocols.

We implement a widely used hash-to-curve algorithm for BLS12 curves in the `arkworks` ecosystem, which can be widely used beyond this work. In addition, our proposed LMF provides a promising avenue for improving the efficiency of proof generation in systems that require variable-length proofs. We have also successfully identified and resolved a bug in the `arkworks` library that ensures reliable use of field emulation in R1CS circuits.

While our system offers significant improvements, we have also recognized some limitations. The proof remains impractical for certain use cases, primarily due to the large memory requirements. In addition, while the LMF data structure reduces the need for expensive proofs, it also increases memory consumption. Future work could explore more efficient pairing algorithms, better proof systems, and SNARK-friendly signature schemes to improve the utility and scalability of our protocol.

In summary, this paper contributes to the field by providing a novel light client protocol that offers both theoretical and practical insights. We hope that this work will lay the foundation for the future development of secure and efficient light client protocols.

Bibliography

[ac22] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[ACP+24] Zaryab Afser, Aaron Chen, Pablo Pettinari, Nicolás Quiroz, Corwin Smith, and Joseph Cook. Light clients. 2024.

[AKMW24] Frederik Armknecht, Ghassan Karame, Malcom Mohamed, and Christiane Weis. Practical light clients for committee-based blockchains, 2024.

[AME19] Cornelius C Agbo, Qusay H Mahmoud, and J Mikael Eklund. Blockchain technology in healthcare: a systematic review. In *Healthcare*, volume 7, page 56. MDPI, 2019.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, pages 315–334. IEEE, 2018.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 111–120, 2013.

[BdM94] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 435–464, Cham, 2018. Springer International Publishing.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive*, 2019.

[BGS+23] Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, and Yupeng Zhang. Introduction to modern snarks, 02 2023.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[BMRS20a] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Paper 2020/352, 2020.

[BMRS20b] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. *Cryptology ePrint Archive*, 2020.

[Bow19] Sean Bowe. Faster subgroup checks for BLS12-381. Cryptology ePrint Archive, Paper 2019/814, 2019.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 276–294, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 499–530, Cham, 2023. Springer Nature Switzerland.

[CBC22] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. Sok: Blockchain light clients. In *International Conference on Financial Cryptography and Data Security*, pages 615–641. Springer, 2022.

[CRTA+24] Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. Analyzing and Benchmarking ZK-Rollups. In Rainer Böhme and Lucianna Kiffer, editors, *6th Conference on Advances in Financial Technologies (AFT 2024)*, volume 316 of *Leibniz International Proceedings in Informatics (LIPIcs)*,

pages 6:1–6:24, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[DCB25] Trisha Datta, Binyi Chen, and Dan Boneh. VerITAS: Verifying Image Transformations at Scale . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 97–97, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[DHMP23] Gérald Doussot, Kevin Henry, Sam Markelon, and Thomas Pornin. R1cs implementation review penumbra labs, 2023.

[Edg15] Ben Edgington. Bls12-381 for the rest of us - hackmd, 2015.

[FHSS+23] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380, August 2023.

[Fou21] Mina Foundation. 2021.

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535. USENIX Association, August 2021.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, pages 305–326. Springer, 2016.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[Hop20] Daira Hopwood. The pasta curves for halo 2 and beyond, 11 2020.

[Hou23] Youssef El Housni. Pairings in rank-1 constraint systems. In Mehdi Tibouchi and XiaoFeng Wang, editors, *Applied Cryptography and Network Security*, pages 339–362, Cham, 2023. Springer Nature Switzerland.

[KPS18] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961, 2018.

[KS24] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. In *Annual International Cryptology Conference*, pages 345–379. Springer, 2024.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.

[LLM+24] Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. A survey on the applications of zero-knowledge proofs, 2024.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[Nak08] Satoshi Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin.org/bitcoin.pdf*, 9:15, 2008.

[NAK+22] Muhammad Hassan Nasir, Junaid Arshad, Muhammad Mubashir Khan, Mahawish Fatima, Khaled Salah, and Raja Jayaraman. Scalable blockchains—a systematic review. *Future generation computer systems*, 126:136–162, 2022.

[NDC+24] Wilson Nguyen, Trisha Datta, Binyi Chen, Nirvan Tyagi, and Dan Boneh. Mangrove: A scalable framework for folding-based snarks. In *Annual International Cryptology Conference*, pages 308–344. Springer, 2024.

[OWWB20] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092. USENIX Association, August 2020.

[pse23] privacy-scaling explorations. Github - privacy-scaling-explorations/sonobe: Experimental folding schemes library, 2023.

[PSG+24] Charalampos Papamanthou, Shravan Srinivasan, Nicolas Gailly, Ismael Hishon-Rezaizadeh, Andrus Salumets, and Stjepan Golemac. Reckle trees: Updatable merkle batch proofs with applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communi-*

cations Security, CCS '24, page 1538–1551, New York, NY, USA, 2024. Association for Computing Machinery.

[RML] Ronny Roland, Ashok Menon, and Mark Logan. Checkpoint verification | sui documentation.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *J. Cryptol.*, 4(3):161–174, January 1991.

[SMG+17] Charlie Shier, Muhammad Izhar Mehar, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Marek Laskowski, and Henry M. Kim. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *SSRN Electronic Journal*, 2017.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*, pages 1–18. Springer, 2008.

[VGS+22] Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. Plumo: An ultralight blockchain client. In Ittay Eyal and Juan Garay, editors, *Financial Cryptography and Data Security*, pages 597–614, Cham, 2022. Springer International Publishing.

[WB19] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the bls12-381 elliptic curve. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):154–179, Aug. 2019.

[WLWC21] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges, 2021.

[WPG+23] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, AFT '22, page 30–46, New York, NY, USA, 2023. Association for Computing Machinery.

[XZC+22] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3003–3017, 2022.

[XZZ+19] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019.

[YW18] Yong Yuan and Fei-Yue Wang. Blockchain and cryptocurrencies: Model, techniques, and applications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(9):1421–1428, 2018.

Appendix A

SNARK Definition

We define completeness, knowledge soundness, non-interactivity, succinctness, and zero-knowledge for a SNARK below [DCB25, PSG+24, BGS+23].

- Completeness: If $(x, w) \in R$, then verification should pass. That is, for all $\lambda \in N$ and all $(x, w) \in R$:

$$\Pr\left[\text{Verify}(\text{vk}, x, \pi) = 1 \;:\; \begin{array}{c} \text{pk, vk} \xleftarrow{\$} \text{Setup}(1^\lambda, R), \\ \pi \leftarrow \text{Prove}(\text{pk}, x, w) \end{array}\right] = 1$$

- Knowledge Soundness: For any relation R, and any PPT adversary A, there exists a PPT extractor E such that the following probability is negligible in λ.

$$\Pr\left[\begin{array}{c} \text{Verify}(\text{vk}, x, \pi) = 1 \\ \wedge (x, w) \notin R \end{array} \;:\; \begin{array}{c} \text{pk, vk} \xleftarrow{\$} \text{Setup}(1^\lambda, R), \\ (x, state) \xleftarrow{\$} A(pk, vk) \\ w \xleftarrow{\$} E(state, \text{pk}, x) \end{array}\right]$$

- Non-interactivity: The proof is non-interactive.

- Succinctness: The proof size is $o(|w|)$ and the verifier can run in $O_\lambda(|x|) + o_\lambda(|C|)$ where $|C|$ is the number of the constraints in the arithmetic circuit.

- Zero-Knowledge: Consider an oracle $H : X \to Y$ for some finite sets X and Y. The zk-SNARK is zero knowledge in the random oracle model if there is a PPT simulator Π such that for all $(x, w) \in R$ and all PPT adversaries A, the

following function is negligible

$$\mathrm{Adv}^{\mathrm{zk}}_{\mathcal{A},\Pi}(\lambda) := \left| \begin{array}{c} \Pr\left[A^{H}((\mathrm{pk},\ \mathrm{vk}), x, \mathrm{Prove}^{H}(pk, x, w)) = 1\right] - \\ \Pr\left[\mathcal{A}^{H[h]}((\mathrm{pk},\ \mathrm{vk}), x, \pi) = 1\right] \end{array} \right|$$

where pk, vk $\xleftarrow{\$}$ Setup$(1^{\lambda}, R)$, $(\pi, h) \xleftarrow{\$} \Pi((pk, vk), x)$, $h : X \to Y$ is a partial function, and $H[h]$ refers to the oracle $H : X \to Y$ modified by entries of h.

Appendix B

Project Plan and Interim Report

B.1 Project Plan

B.1.1 Background

Since the introduction of Bitcoin, blockchain has evolved into a transformative force in a variety of fields and communities. Originally popularized as the backbone of cryptocurrencies [YW18], blockchain has now enabled a wide range of applications, including decentralized finance (DeFi) [WPG+23], non-tamperable tokens (NFT) [WLWC21] and healthcare [AME19]. In simple terms, blockchain is a distributed system where each node keeps a copy of all transactions. When new transactions occur in the network, nodes reach a consensus according to a defined protocol to extend the blockchain.

With the gradual development of blockchain, different types of nodes have started to emerge. They can generally be categorized as follows. **Consensus nodes** are responsible for ensuring the integrity and security of the network by running a consensus protocol to agree on the state of the blockchain. **Full nodes** maintain a complete copy of the blockchain, store all historical data, and communicate with other full nodes through the gossip protocol. These nodes have sufficient resources to handle the storage, bandwidth, and computation requirements associated with maintaining the entire blockchain history. On the other hand, **light clients** are clients with limited resources (e.g., mobile devices or browsers). They do not store the complete blockchain and lack the resources to perform heavy computational tasks. Instead, they rely on full nodes to act as intermediaries, fetching blockchain data and submitting transactions on their behalf [CBC22].

For example, the Simplified Payment Verification (SPV) protocol, as proposed in the Bitcoin whitepaper [Nak08], is a protocol that allows light clients to verify transactions without downloading the full blockchain. Similarly, in Ethereum [ACP+24], there is a light-client protocol that relies on checkpoints and a committee-based mechanism to account for the chain's fast block generation rate. However, although called light client protocols, these protocols may still be too resource intensive for some lightweight environments because light clients still need to download some data that is linearly proportional to the size of the chain (e.g., the chain of block headers in SPV). After downloading the data, the client also needs to verify that the downloaded chain of headers starts from the genesis block and that each block in it has been correctly agreed upon by the consensus nodes, which is also an operation that is linearly proportional to the chain size. This kind of data and operation is too demanding for lightweight clients, considering that many of the latest chains optimize block generation rates [NAK+22] and lightweight clients often go through offline phases [CBC22].

To solve this, the full nodes need to compress the information sent to the client, while still providing the security guarantee that the light client wants. A promising solution is to leverage **SNARK**, which are succinct non-interactive proofs (arguments) of knowledge [BBB+18, GWC19, XZZ+19]. It provides a complete, knowledge-sound and succinct proof system. **Recursive SNARK** [BCCT13], which is a technique that composes SNARK together, enables even more efficient proof generation and verification for Incrementally Verifiable Computation (IVC) [Val08]. It is particularly useful in blockchain (which can be formulated as an IVC problem), utilizing which full nodes can generate succinct and efficient proofs, thus greatly reducing the burden on light clients. This scheme is further improved with a series of recent efforts to design **folding** schemes [BGH19, KST22, KS24]. Using the folding scheme, provers can fold multiple witnesses for the same instance together, and then only generate a proof for the single folded instance, thus reducing the proving time.

Due to its appealing features, SNARK has been deployed in a variety of existing protocols and blockchains. For example, zkEVM [LLM+24] embeds SNARK proofs into Ethereum Virtual Machine (EVM) executions to prove the correctness of execution of transactions in Ethereum. However, this is not a solution to our problem because it focus on proving the correctness of transaction rather than consensus. zkbridge [XZC+22] is a protocol that utilizes SNARK to support for cross-chain transactions. It is more relevant to our problem, but its security is based on the assumption that there exists a consistent, succinct, and live light-client protocol, which is what we want to improve upon, and it only proves one state transition at a time. Compared

with the previous approaches, Mina [BMRS20b] goes a step further. It designs the entire blockchain based on SNARK, which provides proof of security for consensus and transaction execution whenever a block is generated. Thus, in Mina, light clients have the same full security guarantees as full nodes [Fou21]. However, its approach is not directly applicable to our work; our goal is to focus on proving the correctness of consensus and designing a generalized protocol for different chains.

B.1.2 Aims and Objectives

Aims

Thus, the aims of this project are

- Design a secure protocol that enables efficient and succinct verification of the blockchain consensus. Specifically, the focus is on supporting the verification of a chain of signatures, which is often used in blockchain checkpoints and is an important part of light client protocols.
- Make the protocol practical by exploring possible performance optimizations and implementing it on a real blockchain.

Objectives

- Review various blockchains and explore how to programmatically interact with them, in particular how to query the blocks and checkpoints and verify that the consensus is done correctly.
- Define the security properties of the protocol, design it and prove its security.
- Implement different SNARK circuits that instantiate the protocol and are capable of verifying the checkpoints validity for real-world blockchains.
- Experiment and analyze the performance of different circuit implementations and suggest performance optimizations for the circuits and protocols.
- Evaluate the feasibility of deploying the protocol in a real-world blockchain, including providing specific cost analysis.

B.1.3 Expected Outcomes and Deliverables

- A protocol that enables efficient and succinct verification of the real-world blockchain's consensus and a proof that it is secure (with respect to certain properties we want).

- A series of experiments on real blockchains evaluating different SNARK implementations, with analysis and discussion of their performance.
- A feasibility analysis of the proposed protocol on real-world blockchains from cost and performance perspectives.
- Some examples of circuit implementations that can verify certain blockchain checkpoints.

B.1.4 Work Plan

- **07/10/24 to 08/11/24 (5 weeks)**: Review various papers that systematize knowledge of distributed systems and blockchains from different perspectives (e.g., structures, consensus mechanisms, streamlined clients, etc.). Develop project directions by exploring various blockchain scanner platforms, archive node designs, and how cryptographic constructions are used in the latest blockchains (e.g., Verkle Tree, SNARK, and zk-bridge).
- **09/11/24 to 21/12/24 (6 weeks)**: Continue to explore the math behind the folding scheme and how to build a folding-based SNARK. Develop a proof-of-concept for circuits capable of verifying consensus in blockchains like Ethereum and Cosmos. Draft the security proof of the proposed protocol. Write the interim report.
- **22/12/24 to 28/02/25 (11 weeks)**: Complete and submit interim reports. Design and implement various baselines for experiments (monolithic SNARK, recursive SNARK, folding-based SNARK, parallel folding-based SNARK (if possible)). Evaluate the performance of different circuits and think about how to improve the performance of the verifier without compromising protocol security. Perform some concrete cost analysis of the proposed protocol (e.g., how much it costs to generate a proof for each block). Start writing the final report.
- **01/03/25 to 28/03/25 (4 weeks)**: Continue the experiment and the evaluation. Finish final report.
- **01/04/25 to 25/04/25 (4 weeks)**: Iteratively revise the final report and refine some of the analyses and experiments.

B.2 Interim Report

B.2.1 Progress

- Reviewed relevant literature on BLS signatures (e.g., rogue public key attacks), BLS12-381 curves (e.g., curve equations, twist, isogeny, and hash-to-curve algorithms), lattice-based folding schemes, and SNARK applications.
- Learned `arkworks`, a popular library for building SNARK on Rust. Here is a summary of the topics I explored:
- `ark-ff` (`arkworks` representation of finite field): Montgomery modular multiplication and how to define prime and generic fields (prime + extension) in arkworks.
- `ark-ec` (`arkworks` representation of elliptic curve): Montgomery ladder, cofactors, Jacobian coordinates, elliptic curve point serialization and embedding degree.
- `ark-r1cs-std` (`arkworks` representation of R1CS): representation of constraint systems, field variables and field emulation.
- Explored how Sui Blockchain checkpoints work: understanding how committees are represented and how checkpoints and their summary are constructed, signed, and verified.
- Implemented BLS signatures on `arkworks`, including parameter generation, signing, and batch verification.
- Implemented 90% of BLS signatures on `arkworks` R1CS. In the process, I patched hundreds of lines of code in `arkworks` to enable the usage of emulated field variables in the circuit, optimized the circuit to reduce the number of generated constraints by 2/5, and created an implementation of hash-to-field.
- Did some initial testing on AWS EC2 machines.

B.2.2 Remaining Work

- Implement map to curve and cofactor clearing in R1CS to complete the implementation of hashing to curve primitive.
- Fix bug in `arkworks` related to emulated field variable.

- Experiment with constraint generation and use SNARK (e.g. Groth16) to prove BLS signature verification.
- Implement recursive SNARK for committee rotations.
- Explore and implement a folding scheme to speed up the recursive SNARK proof process.

B.2.3 Updated Work Plan

- **24/01/25 to 07/02/25 (2 weeks)**: Complete hash-to-curve implementation. Fix bug in `arkworks`.
- **08/02/25 to 28/02/25 (3 weeks)**: Set up experimental environment (on Sui or in a simulated blockchain). Implement committee rotation validation. Design and implement various baselines (recursive SNARK, folding-based SNARK, parallel folding-based SNARK (if possible)). Evaluate the performance of different circuits. Think about how to improve performance where possible.
- **01/03/25 to 14/03/25 (2 weeks)**: Continue the experiment. Perform concrete cost analysis of the proposed protocol (e.g., how much it costs to generate a proof for each block). Start writing the final report. Finish final report.
- **15/03/25 to 31/03/25 (2 weeks)**: Begin writing and complete the final report.
- **01/04/25 to 25/04/25 (4 weeks)**: Iteratively revise the final report and refine some of the analyses and experiments.

Appendix C

Code and Experiment Data

The code and experimental data are publicly available at `https://github.com/yuxqiu/mim`, with the experiment data located in the `exp` directory.