Efficient Batch Opening Schemes for Merkle Tree Commitment with Applications to Trustless Cross-chain Bridge

Bingsheng Zhang[1], Wuyunsiqin[1], Xun Zhang[1], Markulf Kohlweiss[2,3], Kui Ren[1]

[1]The State Key Laboratory of Blockchain and Data Security, Zhejiang University,
Hangzhou, China, {bingsheng, 3210101763, 22221082, kuiren}@zju.edu.cn
[2]School of Informatics, University of Edinburgh,
Edinburgh, United Kingdom, mkohlwei@ed.ac.uk
[3]Input Output

***Abstract*—In blockchain systems, Merkle trees represent a fundamental cryptographic structure for verifying the validity of public keys in digital signatures. However, the verification process presents significant computational challenges, particularly when dealing with large-scale public key participation in signing operations. This paper focuses on addressing the efficiency bottlenecks in public key validity verification within Merkle tree commitments, with particular emphasis on their application in trustless cross-chain bridge protocols. While existing cross-chain solutions predominantly rely on zero-knowledge proofs for blockchain state validation, the inherent computational cost of proof generation remains prohibitive.**

We present a novel batch opening scheme for Merkle tree commitments that synergistically integrates Merkle tree construction from permutation arguments to verify the membership of extensive leaf sets. Our approach demonstrates remarkable proof generation efficiency advantages, particularly maintaining consistent performance regardless of the number of opened leaves, given a fixed tree depth. Our methods significantly reduce the computational overhead associated with public key validity verification. Meanwhile, it is fully applicable to the existing classical Merkle tree structure without any modifications and has universality.

To demonstrate the practicality and efficiency of our scheme, We implemented the Merkle tree opening circuit for three hash functions (Poseidon, Rescue and Keccak) based on our scheme. Our evaluation shows that the batch opening scheme achieves better performance: proof generation time begins to shorten from an opening ratio of 0.25, achieving a 3.5 to 7.1× improvement at a ratio of 0.75 (with tree depth = 9). Similar improvements are also reflected in the proof size and verification time. Moreover, as tree depth increases, our method's performance advantages become more pronounced.

***Index Terms*—Merkle Tree, Trustless Cross-Chain Bridge, Zero-Knowledge Proof**

I. Introduction

Light clients [1] are specialized nodes within a blockchain network that interact with the system without maintaining the complete blockchain history. When a light client needs to verify a set of transactions, it retrieves the block header at a specific height h. However, without full access to the transaction history, the client is unable to verify the entire chain of block headers on its own.

A common solution to this problem is the use of validators (or signers) who digitally sign the block header to certify its validity. This signature, often implemented as a multisignature or aggregate signature, provides a trust mechanism that assures the light client of the header's authenticity without requiring full verification. Multisignature schemes, such as those in the Schnorr family [2] or pairing-based BLS signatures [3], are capable of aggregating many individual signatures—even those from different keys—into a single compact signature.

In order to verify a multisignature, the verifier constructs an aggregated verification key from t public keys that are committed in a Merkle tree. However, this aggregation process introduces a vulnerability to rogue key attacks [4], where an adversary might forge a multisignature by misrepresenting the associated public keys.

To mitigate this risk, it is essential for the light client to obtain a proof that the aggregated verification key has been correctly assembled. Although one approach is to include the Merkle tree paths from the root to the public keys within the signature, a more concise and efficient solution is to employ zero-knowledge proofs. These proofs enable both the aggregation of Merkle tree proofs and the multisignatures in a manner that is both succinct and secure.

A. Our Contribution

In this paper, we present a novel optimization method for batch opening of Merkle leaves based on permutation argument. Our approach significantly enhances traditional Merkle proof verification methods when a large fraction of the leaves must be verified. The key contributions of our work are as follows:

Batch Opening Schemes for Merkle Tree. We propose an efficient batch opening scheme for Merkle trees. By employing Merkle tree construction and permutation argument, our approach verifies the membership of a large number of leaves in the Merkle tree. Under a fixed number of openings, the scheme

offers significant proof speed advantages, with efficiency that remains independent of the actual number of opened leaves.

Elliptic Curve Encoding. We develop a novel encoding scheme for elliptic curve points that enables efficient permutation verification while preserving the security of the system. The proposed compression technique, in combination with our permutation argument, allows for efficient batch verification without compromising security.

Comprehensive Evaluation. We conducted extensive experiments comparing our protocol with others. Under different hash function settings, our batch opening scheme achieves up to $3.5\times \sim 7.1\times$ improvement in proof generation time at an opening ratio of 0.75. Similar gains are observed in proof size and verification time, with advantages growing as tree depth increases.

B. Applications

Our approach has broad applications in blockchain interoperability, particularly in trustless cross-chain bridges and threshold multisignature validation. Trustless cross-chain bridges, essential for secure asset and data transfers across blockchains, often leverage SNARK-based infrastructures for their succinct proof sizes and fast verification. Our method efficiently handles batch opening of Merkle tree commitments, which is a fundamental step in verifying the validity of a large set of public keys. This is particularly beneficial in threshold multisignature schemes, such as those used in Ethereum 2.0 [5], Cosmos [6], and Cardano [7], where a large number of public keys may be committed in a Merkle tree. By reducing the cost of opening these commitments, our approach enhances the efficiency of verifying public key membership without modifying the underlying signature schemes.

C. Evaluation

Our comprehensive evaluation demonstrates that the proposed SNARK-based approaches deliver significant performance improvements over traditional methods. Extensive experiments using zk-friendly hash functions and various tree configurations reveal that the SNARK-based Merkle Tree proof from permutation argument method consistently achieves lower proof generation times and reduced verification overhead. These advantages are particularly pronounced in scenarios where a large proportion of elements require verification, as is common in practical multisignature and on-chain applications. Overall, our results indicate that, although the optimal method can be selected based on the specific proportion of elements to be opened, the SNARK-based Merkle Tree proof from permutation argument method proves superior in most realistic operational settings.

D. Related work

Directly using SNARKs to construct Merkle proofs and perform batch aggregation suffers from a significant drawback: as the number of proofs to verify increases, the computational cost grows substantially, making it less efficient in large-scale scenarios.

Recent research has explored designing novel vector commitment schemes to address this challenge. [8] introduces zkTree, which recursively verifies child zero-knowledge proofs in a parent node, aggregating multiple proofs into a single root proof for constant on-chain gas cost. [9] presents Hyperproofs, a scheme that combines efficient maintainability with aggregatability; it produces compact, logarithmic-sized algebraic hash proofs and aggregates faster than SNARK-based Merkle Tree proof aggregation. [10] proposes Reckle Trees, embedding batch hash computation within recursive Merkle verification via canonical hashing to achieve logarithmic update time and enable parallel computation. However, practical limitations remain: existing blockchains primarily use Merkle trees, making transitions costly; Hyperproofs may yield larger proofs and slower verification than SNARK-based approaches, while Reckle Trees only offer significant benefits for deep trees, limiting their use in cross-chain applications.

Given these limitations, there remains a need for algorithmic solutions tailored to cross-chain scenarios that can optimize Merkle proof verification while maintaining compatibility with existing blockchain infrastructures and providing efficient on-chain verification capabilities.

II. Preliminary

A. Notation

Let λ be the security parameter and $H : \{0,1\}^* \to \{0,1\}^{2\lambda}$ denote a collision-resistant hash function. For a positive integer n, let $[n] = \{0, 1, \ldots, n-1\}$ represent the index set. A vector $\mathbf{a} = (a_0, \ldots, a_{n-1})$ denotes a sequence of binary strings where $a_i \in \{0,1\}^{2\lambda}$ for all $i \in [n]$. If the length of a_i is arbitrary, the hash function H is used to compress it to a fixed size. For any real number $x \in \mathbb{R}$, the floor function $\lfloor x \rfloor$ is defined as the largest integer $n \in \mathbb{Z}$ satisfying $n \leq x$.

B. Succinct Non-Interactive Arguments of Knowledge

A SNARK (Succinct Non-Interactive Arguments of Knowledge) [11] is a protocol where the prover aims to convince the verifier that they know a witness w such that $(x, w) \in \mathcal{R}$ for a statement x and NP relation $\mathcal{R}$. We will work with arguments of knowledge which assume computationally-bounded provers.

A SNARK is a triple of PPT algorithms $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ defined as follows:

- $\mathsf{srs} \leftarrow \mathsf{Setup}(1^\lambda)$:On input security parameter λ , it outputs a structured reference string srs.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{srs}, x, w)$:On input srs, a statement x and the witness w, it outputs a proof π.
- $0/1 \leftarrow \mathsf{verify}(\mathsf{srs}, x, \pi)$:On input srs, a statement x, and a proof π, it outputs either 1 indicating accepting the statement or 0 for rejecting it.

It satisfies the following properties:

- **Perfect Completeness.** A SNARK protocol Π has perfect completeness if for all $(x, \pi) \in \mathcal{R}$:

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

- **Knowledge Soundness.** For any PPT adversary $\mathcal{A}$, there exists a PPT extractor $\mathcal{X}_{\mathcal{A}}$ such that the following probability is negligible in λ:

$$\Pr\left[\begin{array}{l|l}\mathsf{Verify}(\mathsf{srs}, x, \pi) = 1 & \mathsf{srs} \leftarrow \mathsf{Setup}(1^\lambda); \\ \wedge\, (x, w) \notin \mathcal{R} & (x, \pi; w) \leftarrow \mathcal{A} \parallel \mathcal{X}_{\mathcal{A}}(\mathsf{srs})\end{array}\right].$$

(The notation $(x, \pi; w) \leftarrow \mathcal{A} \parallel \mathcal{X}_{\mathcal{A}}(\mathsf{srs})$ means the following: After the adversary $\mathcal{A}$ outputs (x, π), we can run the extractor $\mathcal{X}_{\mathcal{A}}$ on the adversary's state to output w. The intuition is that if the adversary outputs a verifying proof, then it must know a satisfying witness that can be extracted by looking into the adversary's state.)

- **Succinctness.** For any x and w, the length of the proof π is given by:

$$|\pi| = \mathsf{poly}(\lambda) \cdot \mathsf{polylog}(|x| + |w|).$$

C. Merkle Trees

A Merkle tree [12] is a cryptographic data structure used to compute a succinct and collision-resistant digest C for an underlying dataset $\mathbf{M} := \{m_i \mid i \in [n]\}$, which consists of $n = 2^\ell$ elements (e.g., memory slots, transactions, or public keys). The digest serves as a compact representation of the entire dataset and allows for efficient verification of the correctness of any individual element m_i.

Each element in the dataset $\mathbf{M}$, as well as the output of the hash function H, is assumed to be of size 2λ bits. The number i denotes the index of the element in the dataset, and the value of the element is denoted as value(m_i).

Following is the definition of the Merkle tree construction, Merkle proof, and Merkle proof verification.

Merkle Tree Construction. The Merkle tree is constructed by recursively hashing the elements and internal nodes in pairs until the root hash (digest) is obtained.

Algorithm 1: Merkle.Construct($\mathbf{M}$)

Input: A dataset $\mathbf{M} := \{m_i \mid i \in [n]\}$, where $n = 2^\ell$
Output: Merkle Root C_{root}

```
for each data element m_i ∈ M do
    Compute the hash C_i ← H(i ‖ value(m_i))
Initialize C ← {C_0, C_1, ..., C_{n-1}}
while |C| > 1 do
    for each pair of consecutive nodes (C_{2j}, C_{2j+1}) in C do
        Compute the parent node C'_j ← H(C_{2j} ‖ C_{2j+1})
    Update C ← {C'_0, C'_1, ... C'_j}
The remaining element in C is the Merkle Root: C_root ← C_0
```

Merkle Proof. Given a leaf node m_i and the root digest C_{root}, the Merkle proof $P_i := (S_0, S_1, \ldots, S_{\ell-1})$ is a sequence of sibling hashes along the path from the leaf to the root. Each S_j,where $j \in [\ell]$, is the sibling hash at the corresponding tree level.

Merkle Proof Verification. The verification process uses the Merkle proof to recompute the root hash and checks if it matches the known Merkle root.

Algorithm 2: Merkle.Verify(m_i, P_i, C_{root})

Input: Leaf node m_i with value value(m_i),
index i, Merkle proof $P_i := (S_0, S_1, \ldots, S_{\ell-1})$, and
Merkle root C_{root}
Output: 1 if proof is valid, 0 otherwise

```
Initialize h ← H(i ‖ value(m_i))
for j = 0 to ℓ − 1 do
    Let S_j be the j-th sibling hash in P_i
    if ⌊i/2^j⌋ mod 2 = 1 then
        h ← H(h ‖ S_j)
    else
        h ← H(S_j ‖ h)
return 1 if h = C_root, 0 otherwise
```

D. Polynomial Commitment

A polynomial commitment [13] allows us to compute a short value com for a polynomial f of a potential high degree in a finite field $\mathbb{F}$. Later, one can compute short openings that certify that the polynomial committed by com evaluates to $\beta \in \mathbb{F}$ at some position $\alpha \in \mathbb{F}$. Polynomial commitment should be binding because it should be impossible to open the same point to two different values.

A polynomial commitment over a finite field $\mathbb{F}$ is a tuple of PPT algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ defined as follows:

- $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^d)$: The setup algorithm takes a security parameter λ and degree upper bound d as input, and returns a commitment key ck.
- $\mathsf{com} \leftarrow \mathsf{Commit}(\mathsf{ck}, f)$: The commitment algorithm takes a commitment key ck and a polynomial $f \in \mathbb{F}[X]$ as input, and returns a commitment com.
- $\pi \leftarrow \mathsf{Open}(\mathsf{ck}, f, \alpha, \beta)$: The opening algorithm takes a commitment key ck, a polynomial $f \in \mathbb{F}[X]$, a point $\alpha \in \mathbb{F}$, and a value $\beta \in \mathbb{F}$ as input, and returns an opening proof π.
- $b \leftarrow \mathsf{Verify}(\mathsf{ck}, \mathsf{com}, \pi, \alpha, \beta)$: The verification algorithm takes a commitment key ck, a commitment com, an opening proof π, a point $\alpha \in \mathbb{F}$, and a value $\beta \in \mathbb{F}$ as input, and returns a bit b indicating whether the opening is valid.

A polynomial commitment should be binding, meaning it should be infeasible to open the same commitment com at the same point α to two different values β_1 and β_2.

A polynomial commitment should also be hiding, meaning that the commitment com does not reveal any information about the polynomial itself beyond what is intentionally disclosed. Specifically, given a commitment com to a polynomial

$f(x)$, it should be infeasible for an adversary to extract any information about the coefficients of $f(x)$ or predict its value at points other than the ones explicitly revealed during an opening. This ensures that even if the verifier knows the commitment com, it learns nothing about the underlying polynomial until the prover opens the commitment at specific points.

E. Elliptic Curves

Let E be an ordinary elliptic curve defined over a finite field $\mathbb{F}_p$, where p (with $p > 5$) is a prime. The additive group $E(\mathbb{F}_p)$ consists of points (x, y) satisfying the short Weierstrass equation:

$$E/\mathbb{F}_p : \quad y^2 = x^3 + ax + b,$$

with $x, y \in \mathbb{F}_p$, and a point at infinity $\mathcal{O}_E$.

Let $\#E(\mathbb{F}_p)$ denote the cardinality of $E(\mathbb{F}_p)$. It is well known that

$$\#E(\mathbb{F}_p) = p + 1 - t,$$

where t is the trace of the p-power Frobenius endomorphism $\pi : (x, y) \mapsto (x^p, y^p)$.

III. Merkle commitment with selective opening

A. Definitions

Let $\mathbf{M} = (m_0, m_1, ..., m_{n-1})$ be a sequence of elliptic curve points used to generate the leaf nodes of a Merkle tree, where each m_i is a point on the elliptic curve $E(\mathbb{F}_q)$. The order of the points in $\mathbf{M}$ determines the structure of the Merkle tree. Let $\mathbf{M_I} = (m_{i_0}, m_{i_1}, \ldots, m_{i_{m-1}})$ denote a subset of $\mathbf{M}$, where $m \leq n$, containing the points selected for partial opening. $\mathbf{I} = (i_0, i_1, ..., i_{m-1})$ denotes the indices of the selected points. Let $\mathbf{M_{I'}} = (m_{i'_0}, m_{i'_1}, \ldots, m_{i'_{n-m-1}})$ represent the complement of $\mathbf{M_I}$ within $\mathbf{M}$, i.e., the subset of points not selected for opening. $\mathbf{I'} = (i'_0, i'_1, ..., i'_{n-m-1})$ denotes the indices of the selected points. The Merkle root is defined as: C_{root} = Merkle.Construct($\mathbf{M}$), where the sequence of points in $\mathbf{M}$ determines the Merkle tree structure and the Merkle root.

A permutation function σ reorders the elements of a vector. Let $\mathbf{A} = (a_0, a_1, \ldots, a_{n-1})$ and $\mathbf{A'} = (a'_0, a'_1, \ldots, a'_{n-1})$ be two vectors of the same length. The permutation σ acts on the vector $\mathbf{A'}$ such that:

$$\mathbf{A} = \sigma(\mathbf{A'}) \iff a_i = a'_{\phi(i)}, \forall i \in \{0, 1, \ldots, n-1\},$$

where $\phi : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$ is a bijective function that represents the permutation of indices.

This definition means that the elements of $\mathbf{A'}$ are reordered according to the permutation ϕ to produce $\mathbf{A}$. The vector-level permutation σ acts by rearranging all elements in $\mathbf{A'}$ simultaneously.

Let $f : E(\mathbb{F}_q)^n \times [k] \to \mathcal{C}$ be a function that can be expressed by a arithmetic circuit. Here $E(\mathbb{F}_q)^n$ is the n-dimensional vector space consisting of n elliptic curve points from $E(\mathbb{F}_q)$, and $[k]$ is the indices of the points to be opened. $\mathcal{C}$ is the output space of the function, which could be a set of integers, boolean values, or other encoded representations depending on the circuit design. The function $y = f(\mathbf{M}, \mathbf{I})$ represents the result of applying f to the subset $\mathbf{M_I}$, producing an output in $\mathcal{C}$.

The concatenation of vectors or strings is typically denoted by the symbol $\|$. For example, the concatenation of two vectors $\mathbf{A}$ and $\mathbf{B}$ is represented as $\mathbf{A} \parallel \mathbf{B}$. The length of a vector $\mathbf{A} = (a_0, a_1, \ldots, a_{n-1})$ is denoted by $|\mathbf{A}|$, which is equal to n. Again, the notation $[n]$ represents the set of integers from 0 to $n-1$, i.e., $[n] = \{0, 1, \ldots, n-1\}$. Let $\vec{v}_{[n]}$ denote the ordered vector $(0, 1, \ldots, n-1)$ of length n, representing the identity permutation on n indices.

B. Methods of Merkle Commitment Opening

Strawman Solution. The strawman solution to verify the result of a function f applied to a subset of the dataset is to compute it directly and compare it with the provided result. The relation for this scenario is as follows:

$$\mathcal{R} = \{(x = (\mathbf{M}, y, f), w = (\mathbf{I})) \mid y = f(\mathbf{M}, \mathbf{I})\}.$$

However, this relation is impossible to be applied in the context of SNARKs, as the size of statement grows linearly with the number of elements in the dataset, which is not succinct. Therefore, the verification cost scales linearly and thus is not succinct, making it impractical for use in trustless cross-chain contexts.

SNARK-based Merkle Tree Proof. To lower the proof size and verification cost, an alternative approach is to use a Merkle tree structure and generate a SNARK proof. We organize $\mathbf{M}$ as a Merkle tree and use the Merkle root C_{root} as the commitment to $\mathbf{M}$. We only need to open the required elements in $\mathbf{M}$. This results significantly reduced verification times and more compact proofs. The relation in this case is:

$$\begin{aligned}\mathcal{R} = \{(x = (C_{root}, y, f), w = (\mathbf{M}, \mathbf{I}, \mathbf{P_I})) \mid \\ \forall i \in \mathbf{I},\ \text{Merkle.Verify}(m_i, P_i, C_{root}) = 1 \\ \wedge \quad y = f(\mathbf{M}, \mathbf{I})\},\end{aligned}$$

where $\mathbf{P_I} = \{P_i \mid i \in \mathbf{I}\}$ denotes the set of Merkle proofs corresponding to the opened leaves. Each P_i is a Merkle proof as defined in Section II.C.

We then use SNARK to prove this relation. However, when a significant portion of the elements in $\mathbf{M}$ need to be opened, the cost of the method remains high. When the number of leaves to be opened is proportional to n, the computational cost can be significant, with hash computations reaching $O(n \log n)$ in complexity.

SNARK-based Merkle Tree Proof from Permutation Argument. To lower the proving cost, we propose a more efficient approach that combines the Merkle tree construction from permutation argument. This method proves that the selected set and the unselected set together constitute the original set. Specifically, we demonstrate that the selected set is a subset of the original set and establish the relationship between the

original set and its Merkle commitment C_{root}.The formalized relation for SNARK proofs is as follows:

$$\begin{aligned}\mathcal{R} = \{(x = (C_{root}, y, f), w = (\mathbf{M}, \mathbf{I}, \mathbf{I}', \sigma)) \quad | \\ \text{Merkle.Construct}(\mathbf{M}) = C_{root} \quad \wedge \\ \vec{v}_{[n]} = \sigma(\mathbf{I}||\mathbf{I}') \quad \wedge \quad y = f(\mathbf{M}, \mathbf{I})\}.\end{aligned}$$

This approach results in a more efficient cost of $O(n)$, which remains independent of the number of elements that need to be opened. The permutation argument eliminates the need for individual path verifications, reducing the computational overhead and making the protocol more efficient for large datasets.

C. Encoding Elliptic Curve Points for Permutation

To minimize computational overhead, we adopt a method inspired by Bayer and Groth [14] and the permutation argument proposed in PLONK [15]. This method verifies the permutation by compressing each point on the elliptic curve.

To verify that two sets of points on the elliptic curve, are permutations of each other, we first compress each point. Specifically, for $\mathbf{P} = (p_0, p_1, \ldots, p_{n-1})$ and $\mathbf{P}' = (p'_0, p'_1, \ldots, p'_{n-1})$, where each $p_i = (x_i, y_i) \in E(\mathbb{F}_p)$ and $p'_i = (x'_i, y'_i) \in E(\mathbb{F}_q)$ is a point on the elliptic curve. Let $lsb(y)$ denotes the Least Significant Bit (LSB) of y. The compressed representation of each point thus consists of x and $lsb(y)$.

Treating $\mathbf{P}'$ as a shuffled version of $\mathbf{P}$, we introduce two challenge values, γ and ζ, to define two polynomials based on the compressed representations. The construction proceeds as follows:

1) For the set $\mathbf{P}$, the product is computed as:

$$g = \prod_{i=0}^{n-1} (x_i + \gamma \cdot lsb(y_i) + \zeta).$$

2) For the set $\mathbf{P}'$, the product is computed as:

$$h = \prod_{i=0}^{n-1} (x'_i + \gamma \cdot lsb(y'_i) + \zeta).$$

The sets $\mathbf{P}$ and $\mathbf{P}'$ are considered permutations of each other if the following equality holds:

$$g = h.$$

Theorem 1. Any point $p = (x, y)$ on an elliptic curve $E(\mathbb{F}_q)$, where E is defined by the equation $y^2 = x^3 + ax + b$, can be uniquely determined by x and the least significant bit $lsb(y)$.

Proof. Consider the elliptic curve $E(\mathbb{F}_q)$ defined by the equation:

$$y^2 = x^3 + ax + b,$$

For a given $x \in \mathbb{F}_q$, the right-hand side $x^3 + ax + b$ uniquely determines y^2. The equation $y^2 = x^3 + ax + b$ has at most two solutions y_+ and y_-, corresponding to the positive and negative square roots of y^2. These two solutions satisfy:

$$\begin{aligned} y_+ &= \sqrt{x^3 + ax + b} \mod q, \\ y_- &= -\sqrt{x^3 + ax + b} \mod q. \end{aligned}$$

Due to modular arithmetic in $\mathbb{F}_q$, we have $y_- = q - y_+$. Thus, $y_+ + y_- = q$. Since q is a large prime and therefore odd, the two solutions y_+ and y_- must have opposite parities: One of y_+ or y_- is even ($lsb(y) = 0$), the other is odd ($lsb(y) = 1$).

Given $lsb(y)$, we can unambiguously determine whether the solution is y_+ or y_-. Thus, x determines y^2, and $lsb(y)$ resolves the ambiguity between the two possible values of y. Therefore, the point $p = (x, y)$ on the elliptic curve is uniquely determined by x and $lsb(y)$. □

Theorem 2. Let $\mathbf{P} = (p_0, p_1, \ldots, p_{n-1})$ and $\mathbf{P}' = (p'_0, p'_1, \ldots, p'_{n-1})$ be two sets of points on an elliptic curve $E(\mathbb{F}_q)$, where $p_i = (x_i, y_i)$ and $p'_i = (x'_i, y'_i)$. Randomly choose two challenge values $\gamma, \zeta \in \mathbb{F}$. If

$$\prod_{i=0}^{n-1} (x_i + \gamma \cdot lsb(y_i) + \zeta) = \prod_{i=0}^{n-1} (x'_i + \gamma \cdot lsb(y'_i) + \zeta),$$

then the probability that $\mathbf{P} = \mathbf{P}'$ is at least $1 - \frac{n}{|\mathbb{F}|}$, which is overwhelming.

Proof. By Schwartz-Zippel, the following equality of polynomials holds in $\mathbb{F}[X, Y]$:

$$\prod_{i=0}^{n-1} F_i(X, Y) \equiv \prod_{i=0}^{n-1} G_i(X, Y),$$

where the factors $F_i(X, Y)$ and $G_i(X, Y)$ are defined as:

$$\begin{aligned} F_i(X, Y) &= x_i + lsb(y_i) \cdot X + Y, \\ G_i(X, Y) &= x'_i + lsb(y'_i) \cdot X + Y, \end{aligned}$$

where $i \in [n]$ Since $\mathbb{F}[X, Y]$ is a unique factorization domain, every irreducible factor on the left-hand side must map to an irreducible factor on the right-hand side. Therefore, there exists a one-to-one mapping ϕ, defined as:

$$\phi : \{F_0, \ldots, F_{n-1}\} \to \{G_0, \ldots, G_{n-1}\},$$

such that:

$$F_i(X, Y) = c \cdot \phi(F_i)(X, Y),$$

for some $c \in \mathbb{F}^*$.

Next, we note that the coefficient of Y in both $F_i(X, Y)$ and $G_j(X, Y)$ is 1, and the equality of the products implies $c = 1$ for all i. Hence, the mapping satisfies:

$$F_i(X, Y) \equiv G_j(X, Y),$$

where $\phi(F_i) = G_j$, $i, j \in [n]$. Expanding this equality gives:

$$x_i + lsb(y_i) \cdot X + Y \equiv x'_j + lsb(y'_j) \cdot X + Y.$$

By comparing coefficients of X and Y, we conclude:

$$x_i = x'_j, \quad lsb(y_i) = lsb(y'_j).$$

Thus, $p_i = p'_j$, and since the mapping ϕ is one-to-one, the sets $\mathbf{P}$ and $\mathbf{P}'$ are identical. □

Encoding scheme. Building on the above, in our zero-knowledge proof, we extend the idea by defining three polynomials over the compressed representations of the sets $\mathbf{M_I}$, $\mathbf{M_{I'}}$ and $\mathbf{M}$, where the union of $\mathbf{M_I}$ and $\mathbf{M_{I'}}$ is supposed to be $\mathbf{M}$. Let h, g, and m represent the products computed from these sets, respectively. We compute:

$$h = \prod_{\forall (x,y) \in \mathbf{M_I}} (x + \gamma \cdot lsb(y) + \zeta),$$
$$g = \prod_{\forall (x,y) \in \mathbf{M_{I'}}} (x + \gamma \cdot lsb(y) + \zeta),$$
$$m = \prod_{\forall (x,y) \in \mathbf{M}} (x + \gamma \cdot lsb(y) + \zeta).$$

To verify the permutation, we check the equality:

$$h \cdot g = m.$$

This approach allows us to efficiently verify that the sets $\mathbf{M_I}$ and $\mathbf{M_{I'}}$ together form a valid permutation of the original set $\mathbf{M}$.

IV. Applications

A. Trustless Cross-Chain Bridge

A trustless cross-chain bridge [16] is an essential infrastructure in the interaction between different blockchain networks. These bridges allow for secure and seamless asset and information transfers across various blockchains, which is crucial for the interoperability of the blockchain ecosystem. Recently, many trustless cross-chain bridges are built upon SNARK-based infrastructures, owing to their succinct proof sizes and fast verification times, while eliminating the need to trust any third party.

In many PoS [17] blockchains, the light client [1] validates the validity of a block header by verifying signatures (for example, Ethereum 2.0 [5] confirms the block header via the sync committee's signatures), thus eliminating the need to retrieve the complete transaction data. A trustless cross-chain bridge mimics this light client verification process by encapsulating it into a SNARK proof, enabling efficient and secure cross-chain validation. Furthermore, to ensure the validity of signatures, a minimum threshold of valid signatures must be reached and the authenticity of signers must be guaranteed; consequently, in the context of SNARK-based verification, a substantial number of public keys may need to be revealed.

In this scenario, our approach can be utilized in the cross-chain process for proving the validity of threshold multisignatures.

Threshold Multisignatures. Threshold multisignatures [18] allow a group of signers to collectively produce a single signature that remains valid only if at least a predefined threshold number of participants have signed. This cryptographic primitive is widely used in blockchain applications to enhance security and scalability.

For instance, BLS signatures [3] are employed in Ethereum 2.0 [5] and the Cosmos ecosystem [6] to facilitate efficient and compact validation of aggregated signatures. In the Cardano blockchain [7], Mithril [19] serves as a signature scheme to generate chain-linked cryptographic certificates, enabling lightweight and efficient state verification.

In our protocol, the generic function can accommodate these types of signature schemes, allowing flexible integration with different threshold multisignature mechanisms for secure and trustless cross-chain verification.

Threshold multisignatures require selecting an appropriate scheme based on the predefined threshold, as different schemes offer varying trade-offs in terms of efficiency, security, and signature aggregation. When the number of public keys that need to be revealed is large, the computational and storage overhead can become a significant bottleneck. Our approach provides a substantial advantage in such scenarios by significantly improving the efficiency of batch proving public keys within a SNARK proof. This optimization reduces the proving time, making it particularly suitable for applications where large-scale threshold multisignatures need to be validated in a trustless and efficient manner.

In another extreme case, where only a small subset of public keys is exempt from signing, an alternative relation can be utilized. This approach was proposed in [20], enabling a more efficient verification process by leveraging a different formulation of the proof. In this scenario, our generic function can be instantiated as a public key aggregation function, which optimally handles the verification of the signed subset while maintaining security guarantees. The formal representation of this relation is given as follows:

$$\begin{aligned}\mathcal{R} = \{(x = (C_{root}, AVK_M, AVK_I, f_{agg}),\\ w = (\mathbf{M_{I'}}, \mathbf{I'}, \mathbf{P_{I'}})) \mid\\ AVK_M = f_{agg}((\mathbf{M_{I'}} || AVK_I), [|\mathbf{M_{I'}}| + 1])\\ \wedge \forall i' \in \mathbf{I'},\ \text{Merkle.Verify}(m_{i'}, P_{i'}) = 1\}.\end{aligned}$$

Here AVK_M and AVK_I represent two aggregated public keys, where AVK_M represents the aggregated verification key for all potential signers, and AVK_I is the aggregation of the keys from actual signers. The function f_{agg} serves as the public key aggregation function, which combines multiple public keys into a single representative key. Each element in $\mathbf{M}$ corresponds to an individual public key.

This relation establishes that the union of the signing public keys and the non-signing public keys forms the complete set of public keys. Additionally, it ensures that the non-signing public keys are valid, thereby guaranteeing the validity of the remaining signing public keys.

We will later discuss its performance in the Evaluation section.

V. Evaluation

In this section, we evaluate the performance of our proposed method by discussing the experimental setup, metrics used,

results obtained, and a detailed discussion of the findings. The experiments were conducted on a Ubuntu 20.04.6 LTS server. Our SNARK implementation utilizes the halo2 framework.

A. SNARK-based Merkle Tree Proof vs. Strawman Merkle Tree Proof

We first compare the proof of SNARK-based Merkle tree construction with the proof of strawman Merkle Tree construction. In our implementation, we employ a zk-friendly 256-bit Poseidon hash [21] as the underlying hashing component for constructing the Merkle Tree. We conducted experiments with tree depths ranging from 4 to 12 (satisfying the requirements for cross-chain multisignature verification), as shown in Figure 1.

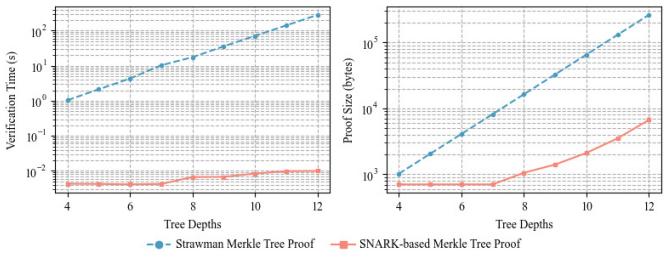

Fig. 1. Performance comparison of SNARK-based Merkle Tree construction proof versus strawman Merkle Tree proof across tree depths from 4 to 12. Using 256-bit Poseidon hash

The communication overhead and verification time of the strawman Merkle Tree proof are significantly higher than those of the SNARK-based proof. Specifically, verification time is at least 20× slower, and the gap widens with increasing tree depth, making the SNARK-based approach more suitable for on-chain scenarios with limited computational resources.

B. SNARK-based Merkle Tree Proof from Permutation Argument vs. SNARK-based Merkle Tree Proof

Next, we compare the SNARK-based Merkle Tree proof method with the SNARK-based Merkle Tree proof from permutation argument method. In both approaches, we employ a 256-bit Poseidon hash as the underlying hash component for constructing the Merkle Tree. The proportion of opened leaf nodes is fixed at 0.75. Experiments were conducted using tree depths ranging from 4 to 12, as illustrated in Figure 2.

With increasing tree depth, differences in proof time, proof size, and verification time become more pronounced. Within the PLONK-based halo2 [22] architecture, the impact of permutation is very small, adding only 1.08s to the overall cost, with the hash function dominating the expense. When the tree depth is below 6, both approaches perform comparably. However, at a tree depth of 9 ($|\mathbf{M}| = 512$), the SNARK-based Merkle Tree proof from permutation argument method requires only 18.27s compared to 63.98s for the SNARK-based Merkle Tree Proof approach—a 3.5× improvement. As tree depth increases further, the performance advantage becomes even more significant.

C. Comparison on Three Different Hash Functions

We also compared the performance of different hash functions under varying open ratios. We tested the performance of the SNARK-based Merkle Tree proof from permutation argument and SNARK-based Merkle Tree Proof methods at open ratios increasing by 0.125 with the tree depth fixed at 9. The experiments employed three hash functions: Poseidon [21], Rescue [23], and Kaccak [24]. The test results are shown in Figure 3.

It can be observed that when the open ratio is at least 0.20, our method consistently achieves shorter proof generation times than the Merkle Proof method across all tested hash functions. Moreover, as the open ratio increases, the performance gap becomes more pronounced. At an open ratio of 0.75, our method outperforms the SNARK-based Merkle Tree Proof method by approximately 3.5×, 7.1×, and 5.3× when using the Poseidon [21], Rescue [23], and Kaccak [24] hash functions, respectively. Notably, more complex hash functions yield greater performance improvements. Additionally, if the complexity of the Merkle leaf information increases, the performance of our method tends to decrease at lower open ratios, whereas its advantages become even more evident at higher open ratios.

D. SNARK-based Merkle Tree Proof from Complement

In Section IV Applications, where we instantiate a multisignature scenario, we discussed that when the vast majority of public keys participate in the signature, verification can be performed using the Merkle proof of the non-signing public keys. To explore the crossover point between our proposed scheme and the SNARK-based Merkle Tree proof from complement method, we conducted tests with a fixed tree depth of 9 and a 256-bit hash function. The experiments varied the open ratio from $\frac{24}{32}$ to $\frac{31}{32}$ in increments of $\frac{1}{32}$. The test results are presented in Figure 4.

It can be observed that when the open ratio is less than or equal to about 0.83, the proof generation time for SNARK-based Merkle Tree proof from permutation argument method is superior to that of the SNARK-based Merkle Tree proof from complement method. Conversely, the SNARK-based Merkle Tree proof from complement method performs better.

Overall, the SNARK-based Merkle Tree proof from permutation argument method achieves the best performance in the intermediate region, allowing for a flexible choice of scheme based on the multisignature threshold.

E. Cost Analysis

Based on the experimental results presented above, we further perform an algorithmic analysis to evaluate the overhead of each approach, thereby facilitating the exploration of the optimal scheme for different multisignature scenarios.

Cost of SNARK-based Merkle Tree Proof. Verifying the inclusion of a single element requires computing the hash of the leaf node and performing an additional $\log n$ hash computations to traverse from the leaf to the root, where $\log n$ represents the depth of the tree. Let l be the cost of

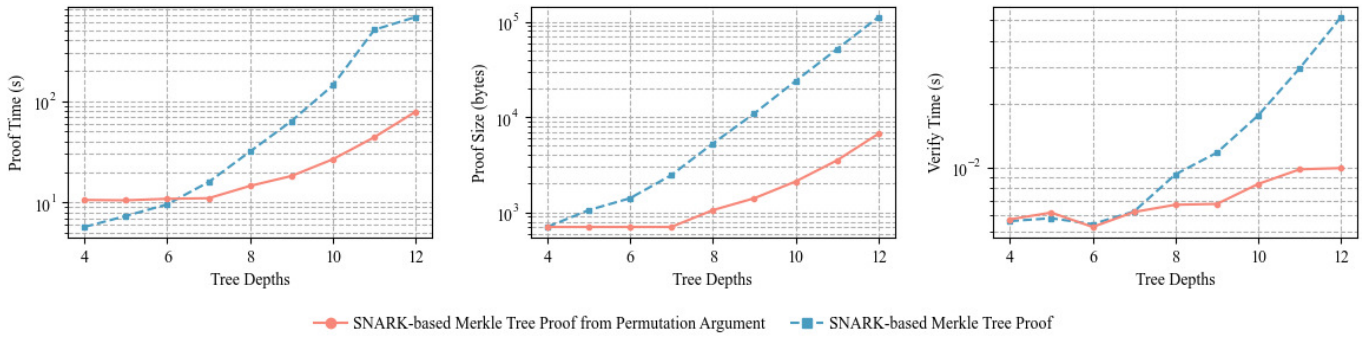

Fig. 2. Comparison of the SNARK-based Merkle Tree proof method and the SNARK-based Merkle Tree proof from permutation argument method using a 256-bit Poseidon hash, with a fixed 0.75 of leaf nodes opened. Results are shown for tree depths ranging from 4 to 12.

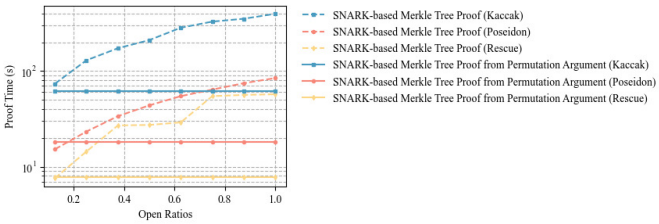

Fig. 3. Performance comparison of SNARK-based Merkle Tree proof from permutation argument and SNARK-based Merkle Tree Proof methods using Poseidon [21], Rescue [23], and Kaccak [24] hash functions across varying open ratios (incremented by 0.125) with a fixed tree depth of 9.

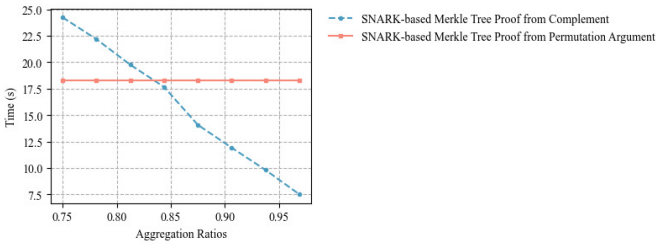

Fig. 4. Comparison between our proposed scheme and the SNARK-based Merkle Tree proof from complement method in a multisignature scenario. Experiments were conducted with a fixed tree depth of 9 using a 256-bit hash, with the open ratio varying from $\frac{24}{32}$ to $\frac{31}{32}$ in increments of $\frac{1}{32}$.

hashing the raw data to the first-level hash. l may vary across different scenarios, with $l = 1$ in our case, assuming $l \ll n$. Thus, verifying a single Merkle proof involves $(l + \log n)$ hash computations. For k selected leaves, the total verification cost amounts to $(k \cdot l + k \cdot \log n)$ hash computations. This complexity scales with both the number of selected leaves k and the dataset size n; specifically, when $k = \Theta(n)$, the overall cost becomes $O(n \log n)$.

Cost of SNARK-based Merkle Tree proof from permutation argument. In contrast, our protocol that leverages a Merkle tree combined with a permutation argument avoids the dependency on the number of selected leaves k. The overall cost includes constructing the Merkle tree for the entire dataset and verifying the permutation. Constructing a Merkle tree for a dataset of n elements requires $(n \cdot l)$ hashes for computing the leaf nodes and an additional $(n-1)$ hashes for building the internal nodes, resulting in a total of $(n \cdot l + n - 1)$ hash computations. The permutation argument then ensures that the selected and unselected leaves together form a valid permutation of the original dataset, which is verified through polynomial evaluations and multiplications. Since this permutation verification operates directly on the dataset, its complexity is linear with respect to n. Consequently, the total cost of this protocol is $O(n)$, making it independent of the number of elements that need to be opened.

Cost of SNARK-based Merkle Tree Proof from Complement. When a significant portion of the dataset needs to be used, the sum of all valid elements is provided, and the unused elements are subtracted, with their validity verified concurrently. The computational cost for verifying the SNARK-based Merkle Tree proof from complement involves generating signatures for the valid elements and performing hash verifications for the unused elements. This approach significantly reduces the overall verification cost, as the overhead becomes proportional to $(n-k) \cdot (l + \log n)$, where k is the number of elements used. As k approaches n, the overhead decreases, leading to a more efficient verification process.

Comparative Analysis. The relative efficiency of the four methods depends on the relationship between k and n. When $k < \frac{(l+1)n}{l+\log n}$, the SNARK Merkle Proof verification is more efficient. Therefore, for smaller values of k, the traditional Merkle Proof verification method is preferable, as its complexity is dominated by $O(k \log n)$. When $\frac{(l+1)n}{l+\log n} \leq k \leq \frac{n(\log n - 1)}{l+\log n}$, the SNARK-based Merkle Tree proof from permutation argument method becomes more efficient. In this range, the overhead of constructing the Merkle tree and verifying the permutation with linear complexity $O(n)$ is more favorable than the growing cost of verifying multiple Merkle Proofs. Finally, when $k > \frac{n(\log n - 1)}{l+\log n}$, the SNARK-based Merkle Tree proof from complement method becomes the most efficient. In summary, the optimal method can be selected based on the actual proportion of elements that need to be opened; however, our proposed SNARK-based Merkle Tree proof from permutation argument method is superior in most practical

scenarios.

VI. Conclusion

In this paper, we have introduced an efficient batch opening scheme for Merkle tree commitments, specifically tailored for cross-chain applications. By combining Merkle tree construction from permutation arguments, our approach enables the verification of a large number of leaves while achieving significant improvements in proof generation speed. Notably, the efficiency of our method remains consistent regardless of the specific number of opened leaves, offering a scalable solution for large-scale verification tasks.

Our comprehensive evaluation demonstrates that the proposed scheme achieves a threefold performance advantage in scenarios where the Merkle tree depth reaches 9 and the opening ratio exceeds two-thirds. These findings underscore the potential of our approach to significantly enhance the scalability and efficiency of light clients operating in trustless cross-chain environments.

Looking ahead, future work could explore further optimizations to the batch verification process, as well as extensions of our approach to other cryptographic primitives within blockchain systems. Additionally, integrating our scheme with emerging consensus protocols could yield even greater improvements in performance and security, ultimately contributing to the development of more robust and interoperable cross-chain solutions.

VII. Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 62232002). This project is supported by Input Output (iohk.io).

Wuyunsiqin is the corresponding author.

References

[1] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "Sok: Blockchain light clients," in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 615–641.

[2] C. Schnorr, "Efficient identification and signatures for smart cards," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 435. Springer, 1989, pp. 239–252.

[3] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International conference on the theory and application of cryptology and information security*. Springer, 2001, pp. 514–532.

[4] T. Ristenpart and S. Yilek, "The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks," in *Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26*. Springer, 2007, pp. 228–245.

[5] Eth2Book, "Altair: Part 2 - building blocks - signatures," 2023. [Online]. Available: https://eth2book.info/altair/part2/buildingblocks/signatures/

[6] "Cosmos," https://cosmos.network/, accessed: 2025-3-12.

[7] "Cardano," https://www.cardano.org/, accessed: 2025-3-12.

[8] S. Deng and B. Du, "zktree: A zero-knowledge recursion tree with zkp membership proofs," *Cryptology ePrint Archive*, 2023.

[9] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang, "Hyperproofs: Aggregating and maintaining proofs in vector commitments," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3001–3018.

[10] C. Papamanthou, S. Srinivasan, N. Gailly, I. Hishon-Rezaizadeh, A. Salumets, and S. Golemac, "Reckle trees: Updatable merkle batch proofs with applications," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1538–1551.

[11] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 2016, pp. 305–326.

[12] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[13] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 6477. Springer, 2010, pp. 177–194.

[14] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer, 2012, pp. 263–280.

[15] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[16] S. Lee, A. Murashkin, M. Derka, and J. Gorzny, "Sok: Not quite water under the bridge: Review of cross-chain bridge hacks," in *ICBC*. IEEE, 2023, pp. 1–14.

[17] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO (1)*, ser. Lecture Notes in Computer Science, vol. 10401. Springer, 2017, pp. 357–388.

[18] P. Gazi, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 139–156.

[19] P. Chaidos and A. Kiayias, "Mithril: Stake-based threshold multisignatures," in *Cryptology and Network Security - 23rd International Conference, CANS 2024, Cambridge, UK, September 24-27, 2024, Proceedings, Part I*, ser. Lecture Notes in Computer Science, M. Kohlweiss, R. D. Pietro, and A. R. Beresford, Eds., vol. 14905. Springer, 2024, pp. 239–263.

[20] P. Gaži, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 139–156.

[21] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *USENIX Security Symposium*. USENIX Association, 2021, pp. 519–535.

[22] Zcash, "Halo2," https://github.com/zcash/halo2, accessed: 2025-3-12.

[23] A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec, "Design of symmetric-key primitives for advanced cryptographic protocols," *IACR Trans. Symmetric Cryptol.*, vol. 2020, no. 3, pp. 1–45, 2020.

[24] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The Keccak Reference." Keccak Team, 2011, accessed: 2025-03-12. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf