zkTree: A Zero-Knowledge Recursion Tree with ZKP Membership Proofs

Sai Deng*and Bo Du†

Polymer Labs

April 15, 2023

Abstract

We introduce zkTree, a general framework for constructing a tree by recursively verifying children's zero-knowledge proofs (ZKPs) in a parent ZKP node, while enabling the retrieval of membership proofs for user-supplied zk proofs. We also outline a construction pipeline that allows zkTree to be built and verified on-chain with constant gas cost and low data processing pipeline overhead. By aggregating a large number of user proofs into a single root proof, zkTree makes ZKP on-chain verification cost-effective. Once the root proof is verified, all user proofs can be verified by providing Merkle membership proofs.

zkTree can be implemented using Plonky2 [34], which combines PLONK [24] and FRI [17], with its root proof recursively verified in Groth16 [27].

Furthermore, we demonstrate how to employ zkTree to verify the default signature scheme of Tendermint [31] consensus by validating ed25519 signatures [5] in a single proof within the Ethereum Virtual Machine (EVM).

1 Introduction

Zero-knowledge proofs are a powerful tool for protecting user privacy and are widely used in blockchains to verify the validity of private transactions, such as Zcash [35]. Another significant application of zero-knowledge proofs is computation compression, where a short on-chain verification can prove that a lengthy off-chain computation has been performed correctly. This verification process requires less time and gas than executing the original computation on-chain. zkEVMs [8] [11], zkRollups [28][13], and zkBridges [36] are examples of such applications.

However, zero-knowledge provers are known for their slow performance. Typically, the time complexity of a prover is at least linear in the size of the arithmetic circuit. In practice, algorithms that are fast on CPUs are not always easily expressed as zk arithmetic circuits. For instance, the widely used EdDSA digital signature scheme over curve25519 [5] requires more than 2 million gates in a zk circuit and a 12-second proving time [3]. Additionally, on-chain verification costs are high, especially on Ethereum (ETH), where the cheapest zk verifier costs around 230k gas [29] and up to 5m gas for a STARK verification [2]. Consequently, many innovative applications, such as zkBridge and zkIBC [32], cannot be deployed due to extended prover proving times and expensive gas costs for on-chain verification.

Our work's primary contribution is the introduction of the zkTree structure and the prototyping of the zkTree recursive proving pipeline to enhance prover performance and reduce verification costs. By distributing proof generation across different machines and recursively composing proofs through zkTree, the prover can achieve near-unbounded computation power and rapid proving speeds. Moreover, by sharing the same on-chain verifier with ZKP membership proofs, different systems or companies can share the invariant on-chain verification costs. By saving time and reducing expenses, zkTree opens up new possibilities for future zk development.

*Email: dengsai@gmail.com; Corresponding author

†Email: bo@polymerlabs.org

2 Background

2.1 Zero-Knowledge Proofs

A non-interactive zero-knowledge system consists of a prover P and a verifier V. The prover aims to demonstrate that they executed a computation C with some public input x and some secret input w, which we call a witness. The prover can send a proof π generated from a public trusted arithmetic circuit implementing C: $P(x, w) \to \pi$. The verifier then verifies the proof with $V(x, \pi) \to$ true/false, without needing to know w. When the computation complexity of $|C| > |V|$, the computation is compressed from the verifier's perspective. The proving work can be moved off-chain, and the on-chain verifier only needs to verify a short proof. This paper focuses on reducing the running time of P and maintaining the complexity and cost of V at a low level.

Numerous zero-knowledge proof protocols exist [27] [24] [33] [30], and they differ in various properties such as trust minimization, security assumptions, proving time, and verification time. Two of the most compelling zero-knowledge technologies on the market today are zk-STARKs [18] and zk-SNARKs [20]. Zk-STARK stands for zero-knowledge scalable transparent argument of knowledge, and zk-SNARK stands for zero-knowledge succinct non-interactive argument of knowledge. At their core, zk-SNARKs depend on elliptic curves for their security, while zk-STARKs rely on collision-resistant hash functions. Consequently, zk-STARKs do not require an initial trusted setup and achieve quantum resistance. However, zk-STARKs have significantly larger proof sizes than zk-SNARKs, resulting in longer verification times and higher gas costs [23].

2.2 Recursive ZKP

One of the latest advancements in efficient zk proof generation is recursive proofs. A recursive zk proof is a proof that verifies some zk proofs inside of its circuit. The prover proves that they verified some inner proofs $P(x_1, \pi_1, x_2, \pi_2, ...) \to \pi$. The proving circuit implements the constraints of zk proof verifiers for the inner proofs. When the verifier verifies the outer proof π, the inner proofs $\pi_1, \pi_2, ...$ are also verified. $V(x, \pi) \to \text{true} \Rightarrow \pi_1, \pi_2, ...$ are true.

Recursive proofs offer the significant advantage of enabling parallel proof generation. This allows the total proving work to be distributed among multiple computers, rather than relying on a single device. Such an approach results in considerable performance improvements for proving multiple circuits or a single circuit that can be divided into small parts. Since the prover carries out work proportional to the circuit size, breaking a large circuit into smaller components will yield performance gains.

Recursive proof composition was first introduced in [21] and later realized in practice using cycles of elliptic curves [19]. Subsequent research, such as Halo [22] and Nova [30], has continued to enhance recursion speed and verification cost. Plonky2 [34], the most recent implementation, employs techniques from PLONK [24] and FRI [17]. It requires only 300ms to generate a recursive proof on a 2021 Macbook Air [34]. In our work, we utilize Plonky2 to implement zkTree due to its rapid recursion speed.

Tree structures are frequently employed in recursive ZKPs to generate the final proof that is verified on-chain. The concept of sharing verification costs by recursively verifying proofs within a tree was first introduced by [10]. zkEVMs [13] also utilize recursion trees for their specific use cases. Our work further improves this approach by enabling the inclusion of heterogeneous zk proofs within a single zk tree.

2.3 Tendermint Light Client

A light client is an application that tracks the consensus state of a blockchain without maintaining complete state [7]. Light clients form the state layer of IBC [25] allow different blockchain protocols to communicate with each other without needing trusted third parties. Recursive zk technology can potentially reduce the on-chain computation costs of executing light client logic. It does this by distributing the cost of verifying different light clients on-chain through recursive proof composition. This will lead to higher throughput, greater speed, and reduced costs for state verification at scale.

Without zk technology, this wouldn't be possible on chains like ETH. For example, Tendermint is a Byzantine Fault Tolerant consensus algorithm [31]. Verifying only the Tendermint light client logic on ETH can exceed the block gas limit [14]. With zk technology, the light client computation can be

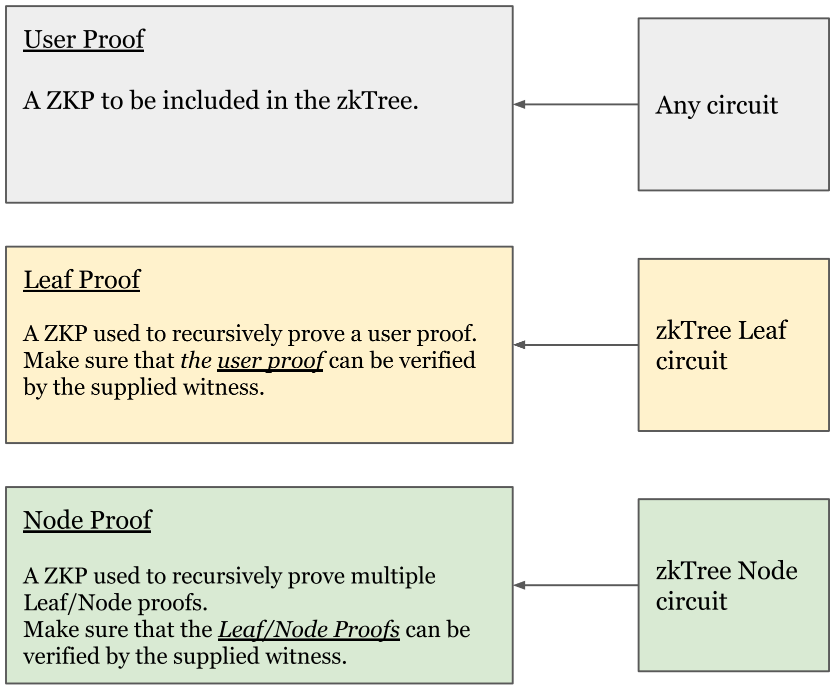
User Proof
A ZKP to be included in the zkTree.
Any circuit
Leaf Proof
A ZKP used to recursively prove a user proof. Make sure that the user proof can be verified by the supplied witness.
zkTree Leaf circuit
Node Proof
A ZKP used to recursively prove multiple Leaf/Node proofs. Make sure that the Leaf/Node Proofs can be verified by the supplied witness.
zkTree Node circuit

Figure 1: zkTree proof types.

moved off-chain. Additionally, using recursive zk technology, multiple light client computations can be composed into one simple zk proof. This allows for the ability to verify the consensus of different blockchains with constant gas costs [29]. In this paper, we demonstrate how to utilize zkTree to verify ed25519, the default signature scheme of Tendermint, on-chain in the EVM.

3 zkTree

zkTree is a tree data structure in which each node represents a zk proof, and every parent node recursively verifies its children's zk proofs. Employing zkTree to prove multiple proofs within a single proof allows for a significant distribution of on-chain verification costs compared to verifying proofs individually. When the root node of zkTree is verified on-chain, all included proofs are also verified on-chain. Figure 2 depicts an example zkTree proof that recursively verifies four user proofs.

A non-interactive proof system comprises a triple (S, P, V). Here, V represents the verifier, P the prover, and S the set of public parameters or common reference strings. These strings typically consist of two parts: verifier data VD and prover data PD. The set S is calculated by preprocessing the public arithmetic circuits. The public inputs of the proof are denoted by x. For the sake of simplicity, private inputs are not discussed in this paper.

There are three types of proofs in a zkTree, as depicted in Figure 1: the user proof π, leaf proof υ, and node proof ω. User proof π_i may be generated from various circuits using different zk schemes and configurations. Each π_i is associated with a unique VD_i. Distinct υ_i proofs are produced by different Leaf circuits of the same zk scheme, while separate ω_i proofs are generated by the same Node circuit in the same zk scheme as υ_i. A zkTree is mathematically constructed using the sets π, υ, and ω, subject to the following constraints:

User i executes

$$P_i(\{x_i\}) \rightarrow \pi_i, \mathrm{VD}_i$$

to generate the user proof π_i for inclusion in a zkTree with public inputs $\{x_i\}$ and verifier data VD_i.

The zkTree Leaf builder runs

$$L_i(\pi_i, \{x_i\}, \mathrm{VD}_i) \rightarrow \upsilon_i, h_i, c_i$$

to verify π_i with other inputs and generate the leaf proof υ_i. Here $c_i = H(\mathrm{VD}_l || \mathrm{VD}_i)$, VD_l is the hash of verifier data of the Leaf circuit, and $h_i = H(\{x_i\})$ is the hash of all the user proof's public inputs.

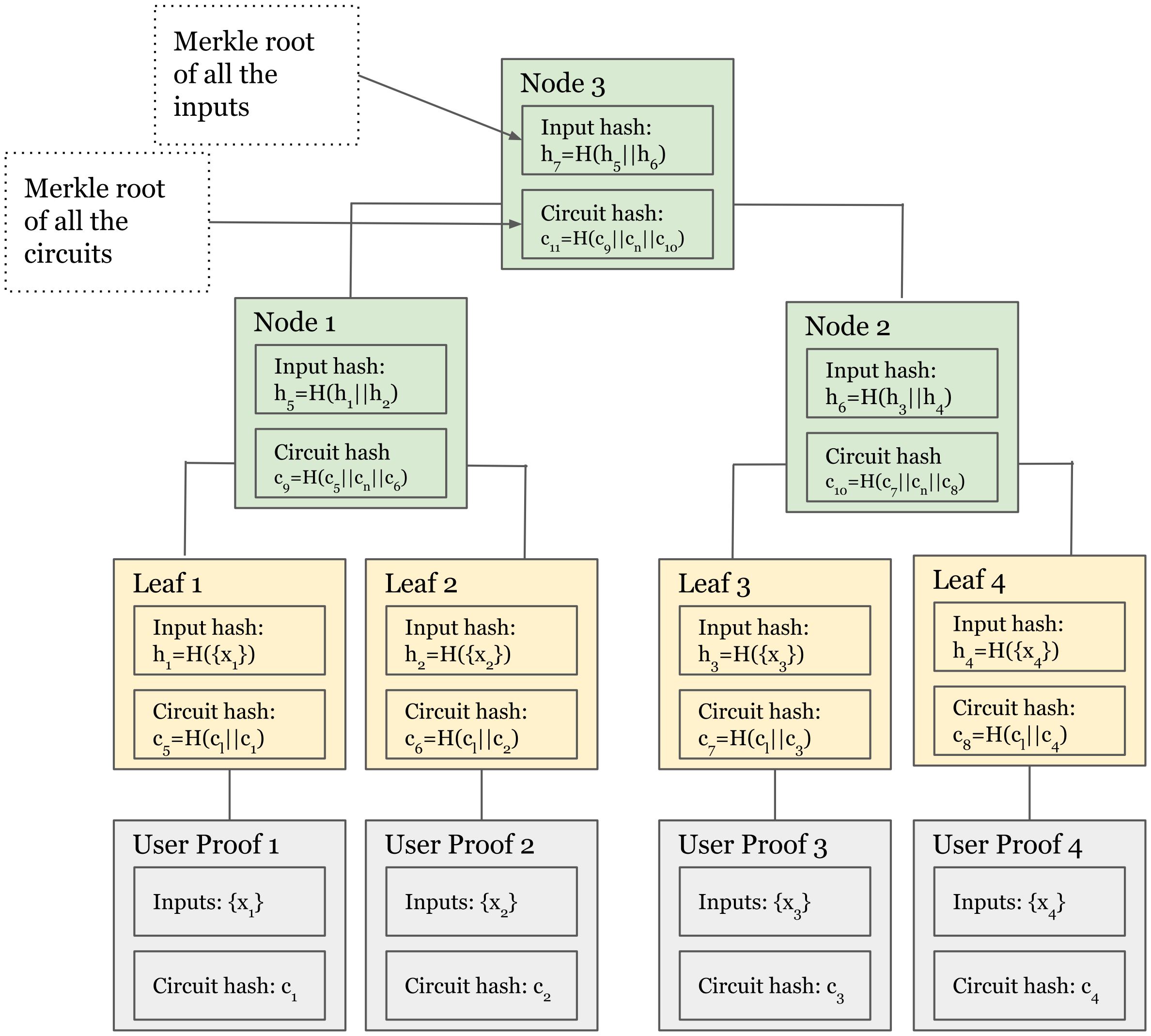
Merkle root of all the inputs
Merkle root of all the circuits
Node 3
Input hash: h_7=H(h_5||h_6)
Circuit hash: c_{11}=H(c_9||c_n||c_{10})
Node 1
Input hash: h_5=H(h_1||h_2)
Circuit hash c_9=H(c_5||c_n||c_6)
Node 2
Input hash: h_6=H(h_3||h_4)
Circuit hash c_{10}=H(c_7||c_n||c_8)
Leaf 1
Input hash: h_1=H({x_1})
Circuit hash: c_5=H(c_l||c_1)
Leaf 2
Input hash: h_2=H({x_2})
Circuit hash: c_6=H(c_l||c_2)
Leaf 3
Input hash: h_3=H({x_3})
Circuit hash: c_7=H(c_l||c_3)
Leaf 4
Input hash: h_4=H({x_4})
Circuit hash: c_8=H(c_l||c_4)
User Proof 1
Inputs: {x_1}
Circuit hash: c_1
User Proof 2
Inputs: {x_2}
Circuit hash: c_2
User Proof 3
Inputs: {x_3}
Circuit hash: c_3
User Proof 4
Inputs: {x_4}
Circuit hash: c_4

Figure 2: A zkTree example with four user proofs.

To construct a node, the zkTree node builder can use two leaves, one leaf and one node or two nodes as the inputs:

$$N(\{\upsilon_i, h_i, c_i\}, \{\upsilon_j, h_j, c_j\}) \rightarrow \omega_k, h_k, c_k$$

$$N(\{\upsilon_i, h_i, c_i\}, \{\omega_j, h_j, c_j\}) \rightarrow \omega_k, h_k, c_k$$

$$N(\{\omega_i, h_i, c_i\}, \{\omega_j, h_j, c_j\}) \rightarrow \omega_k, h_k, c_k$$

Circuit N verifies two input proofs in the node circuit. $h_k = H(h_i||h_j)$ and $c_k = H(c_i||\mathrm{VD}_n||c_j)$, where VD_n is the verifier data of the Node circuit.

When implementing zkTree using Plonky2, the hash of the verifier data can be replaced with the circuit hash, which includes an encoding of gate constraints. The circuit hash and input hash computed in the root node correspond to the Merkle roots of all circuit hashes and public inputs. To verify if a user proof is included in the root proof, it is only necessary to validate the Merkle path of its input hash and circuit hash. For example, as depicted in Figure 2, to confirm that user proof 4 is included in root proof Node 3, the circuit hashes c_4, c_7, c_9 and the input hashes h_4, h_3, h_5 must be provided. c_l and c_n represent the circuit hashes of the leaf circuit and node circuit, respectively. As public parameters, they are used to verify the security of the zkTree builder circuits.

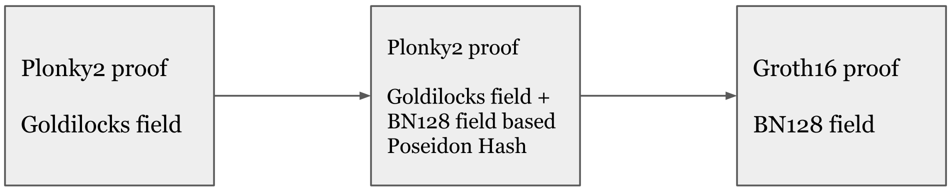

Figure 3: zkTree ETH verifier.

4 On Chain Verifier

Plonky2 proofs are costly to verify on-chain within the EVM. Two steps in the Plonky2 verifier that are particularly expensive include the evaluation of PLONK custom gate constraints and the verification of the FRI protocol. Both steps necessitate heavy computations, such as arithmetic operations over the Goldilocks field [34] and hashing. Even with the substitution of the hash function with an EVM-friendly hash function like keccak256, which has a special EVM opcode with a 36 gas cost per operation, the total gas costs remain infeasible. For instance, verifying a 50KB Plonky2 proof in the EVM with keccak256 hashing incurs approximately 18M gas [12].

To mitigate this cost, we suggest a method to recursively prove the zkTree root Plonky2 proof into a Groth16 proof, which only requires 230k gas with the precompiled contract of pairing checks in the EVM [6].

For recursively proving a ZKP, the complexity is the sum of prover and verifier complexities. In FRI-based protocols like Plonky2, hashing presents the main bottleneck for both the prover and verifier [17]. Plonky2 employs Goldilocks-based Poseidon hashing [26], with about 75% of the recursive circuit dedicated to hashing operations for verifying Merkle proofs [34].

Groth16, PLONK, or other KZG-based zk-SNARK schemes [29] require a pairing-friendly prime field. Since the Goldilocks field is not pairing-friendly, it is necessary to implement the Goldilocks-based circuit in a different field, such as bn128 or bls12-381. Non-native field operations entail expensive range checks and can constitute a significant portion of the recursive circuit. To optimize the Groth16 circuit and reduce overall proving time, we introduce an intermediate Plonky2 proof utilizing a Poseidon-based hash function over a pairing-friendly field within the Plonky2 prover. The Groth16 circuit size is optimized by eliminating numerous non-native range checks when executing hashing in the FRI protocol verification. For example, in the ETH zkTree verifier shown in Figure 3, an intermediate Plonky2 proof using a bn128 field-based Poseidon hash function serves as the middle recursive proof.

5 Distributed Proof Generation

Unlike the approach in [36], which relies on circuit splitting and proof aggregation, zkTree features independent proofs at the same level, eliminating the need for communication costs to generate same-level proofs. Theoretically, zkTree generation time is equal to $log(n)$ times the time required to prove a node proof, plus the time needed for transmission of a node proof between workers. Here, n represents the number of user proofs. The total communication cost is n times the size of a proof. For a Plonky2-based zkTree construction, the size of a proof is approximately 130 KB.

The zkTree generation process is highly adaptable, allowing for extensive customization in production. Depending on the hardware configuration of the workers, multiple nodes can be generated by a single worker. As illustrated in Figure 4, worker 1 may have a lower computational load than worker 2. This heterogeneous load distribution helps reduce communication costs. Additionally, the zkTree ingestion pipeline can be implemented using a data streaming engine. When a high-priority user proof arrives, it can be promptly incorporated into a zkTree construction at a shallower depth compared to other user proofs.

For further acceleration, the Groth16 circuit used to verify Plonky2 proof can be divided into multiple sub-circuits. For instance, each query round in FRI could be placed into a single circuit, while the hash of the public inputs and outputs of each sub-circuit must be verified on-chain to ensure they originate from the same Plonky2 proof. By splitting the Groth16 circuit, a trade-off is made between prover speed and on-chain verification cost. Verification costs increase with the number of sub-circuits.

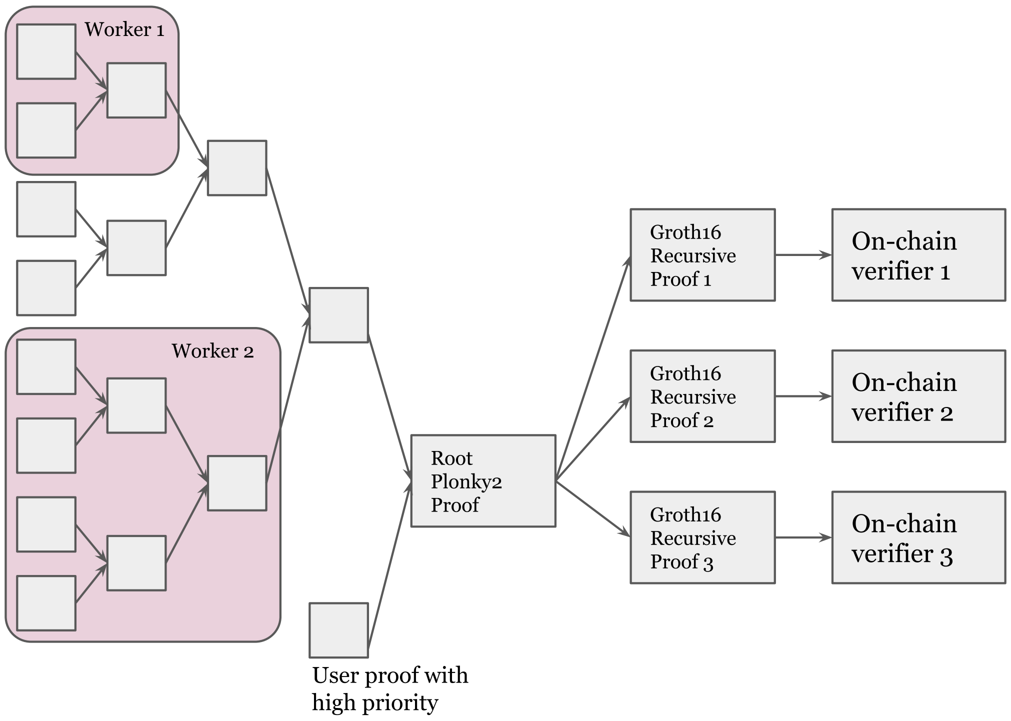
Worker 1
Worker 2
Root Plonky2 Proof
Groth16 Recursive Proof 1
Groth16 Recursive Proof 2
Groth16 Recursive Proof 3
On-chain verifier 1
On-chain verifier 2
On-chain verifier 3
User proof with high priority

Figure 4: zkTree generation.

Stage	Time	Machine Used
Ed25519 proof generation	17s	32 64-core 64GB VM
zkTree generation	5s	16 64-core 64GB VM
Recursive bn128 poseidon based proof generation	9s	1 32-core 256GB VM
Recursive Groth16 witness generation	10s	1 32-core 256GB VM
Recursive Groth16 proof generation	36s	1 32-core 256GB VM

Table 1: Time used for different stages of the zkTree pipeline.

6 Results

We implemented zkTree using Plonky2 and developed an ingestion pipeline with Apache Beam [1]. The Groth16 implementation of the Plonky2 verifier, used for proving the root zkTree proof, is implemented in Circom utilizing rapidsnark [9].

To showcase the practicality of zkTree, we created a prototype for verifying ed25519 signatures and compared the results with other state-of-the-art proving systems. In a Tendermint-based blockchain, such as Cosmos, each block header contains approximately 128 EdDSA signatures (using SHA-512 and Curve25519), with 32 top signatures required to achieve super-majority stakes [15]. To verify Ed25519 proofs on Ethereum, we must simulate curve25519 on curve bn128, resulting in large circuits and extended prover times. However, zkTree enables the distribution of Ed25519 proof generation across multiple machines and the aggregation of these proofs using zkTree.

For zkTree proof generation, we used a test machine, the Google Cloud n2d-highcpu-64, equipped with 64 vCPUs and 64GB RAM. The machine responsible for running Circom witness generation and Rapidsnark prover was a Google Cloud n2-highmem-32, featuring 32 vCPUs and 256GB RAM [4].

In Table 1, we present the time required for each stage of the pipeline. As discussed in the previous section, when the Groth16 recursive circuit is divided, the prover time decreases while on-chain verification costs increase. The balance between cost and time can be adjusted according to the specific needs of a use case, as demonstrated in Figure 5. During the ed25519 proof generation stage, the Plonky2 ed25519 circuit can be partitioned into sub-circuits for inclusion in zkTree, thus further reducing the time.

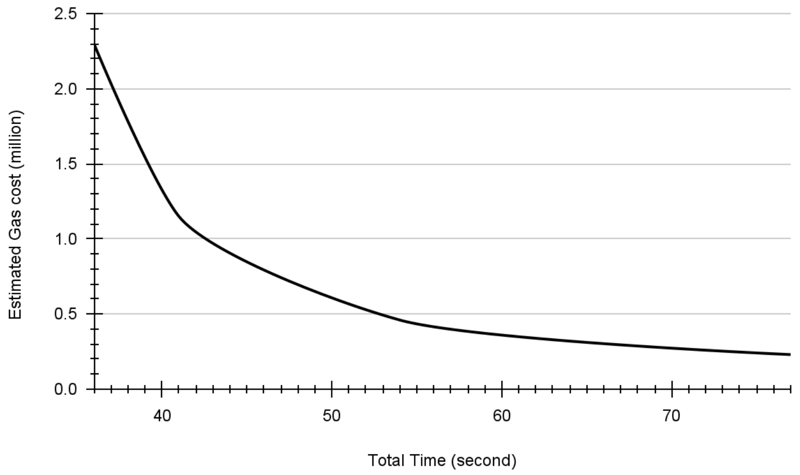

Figure 5: zkTree runtime and cost estimation of verifying 32 ed25519 on ETH.

Method	Time	Machine Used	Communication	Cost (gas)
zkTree	77s	32 64-core VM + 1 32-core VM	<0.1GB	230K
deVirgo	18s	32 96-core VM	32.24GB	230K
Circom-Ed25519[3]	12s	32 16-core VM	<0.1GB	7.36M
Circom-Ed25519	96s	8 16-core VM	<0.1GB	1.64M
Circom-Ed25519	384s	1 16-core VM	<0.1GB	230K

Table 2: Comparison of different methods of verifying 32 ed25519 on ETH.

In Table 2, we compare the speed and cost of several state-of-the-art systems for generating 32 ed25519 proofs. All of these methods use Groth16 for the final on-chain verification proof. It is important to note that the verification cost of zkTree can be significantly reduced by sharing the cost with other systems through the inclusion of more proofs in the tree. Although deVirgo [36] has the lowest total runtime, it depends on a central machine for communication between ordinary machines, making it more susceptible to single points of failure. As the deVirgo circuits grow in size, for example, aggregating 5,000 proofs, the communication cost would be around 5TB. Network bandwidth and the primary machine's hardware would become limiting factors. Finally, since deVirgo is based on circuit-splitting, it lacks the flexibility of zkTree. Any changes to the circuit would necessitate redeploying the updated sub-circuits to every machine.

7 Conclusion and Future Work

zkTree facilitates fast and cost-effective recursive composition of zk proofs. Thousands of ZKPs can be recursively composed and verified on-chain using Merkle membership proofs in approximately one minute and at a cost of 230k gas within a single Groth16 proof. zkTree is flexible, allowing for cost and speed adjustments based on varying use case requirements. The Groth16 proof in the final step can be substituted with a PLONK proof, which does not necessitate a per-circuit trusted setup. By incorporating custom gates, the immediate proof in Figure 3 can be eliminated, allowing for direct verification of the Plonky2 proof in a PLONK proof in less time.

Employing hardware acceleration techniques such as FPGA and ASIC can further enhance the performance of the Plonky2 and Groth16 provers [16]. As a result, the overall time required for zkTree construction and Groth16 proof recursion could be further optimized in the future.

References

[1] *Apache beam.* https://github.com/apache/beam.

[2] *Checkpoints for faster finality in StarkNet - layer 2.* https://ethresear.ch/t/checkpoints-for-faster-finality-in-starknet/9633.

[3] *Circom ed25519.* https://github.com/Electron-Labs/ed25519-circom.

[4] *Compute engine general-purpose machine family | compute engine documentation.* https://cloud.google.com/compute/docs/general-purpose-machines.

[5] *Ed25519: high-speed high-security signatures.* https://ed25519.cr.yp.to/.

[6] *EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128.* https://eips.ethereum.org/EIPS/eip-197.

[7] *Light client.* https://geth.ethereum.org/docs/fundamentals/les.

[8] *Polygon zkEVM.* https://polygon.technology/solutions/polygon-zkevm.

[9] *rapidsnark.* https://github.com/iden3/rapidsnark.

[10] *Reducing the verification cost of a SNARK through hierarchical aggregation - zk-s[nt]arks.*

[11] *Scroll -a native zkEVM layer 2 solution for ethereum.* https://scroll.io/.

[12] *Solidity verifier for plonky2.* https://github.com/polymerdao/plonky2-solidity-verifier.

[13] *Starkware.* https://starkware.co/.

[14] *tendermint-sol.* https://github.com/ChorusOne/tendermint-sol/blob/main/README.md#vanilla-client-branch-main.

[15] *validators-stats.* https://cosmoscan.net/cosmos/validators-stats.

[16] *Zero knowledge proof – InAccel.* https://inaccel.com/zkp/.

[17] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, *Fast reed-solomon interactive oracle proofs of proximity*, in 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018), I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, eds., vol. 107 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 14:1–14:17. ISSN: 1868-8969.

[18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, *Scalable, transparent, and post-quantum secure computational integrity.* https://eprint.iacr.org/2018/046.

[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, *Scalable zero knowledge via cycles of elliptic curves*, 79, pp. 1102–1160.

[20] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, *From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again*, in Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 326–349.

[21] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, *Recursive composition and bootstrapping for SNARKs and proof-carrying data.* https://eprint.iacr.org/2012/095.

[22] S. Bowe, J. Grigg, and D. Hopwood, *Recursive proof composition without a trusted setup.* https://eprint.iacr.org/2019/1021.

[23] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, *A review of zk-SNARKs.* http://arxiv.org/abs/2202.06877.

[24] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, *PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge.* https://eprint.iacr.org/2019/953.

[25] C. Goes, *The interblockchain communication protocol: An overview.* https://arxiv.org/abs/2006.15918v1.

[26] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, *Poseidon: A new hash function for zero-knowledge proof systems.* https://eprint.iacr.org/2019/458.

[27] J. Groth, *On the size of pairing-based non-interactive arguments.* https://eprint.iacr.org/2016/260.

[28] https://matter labs.io, *zkSync — accelerating the mass adoption of crypto for personal sovereignty.* https://zksync.io/.

[29] A. Kate, G. M. Zaverucha, and I. Goldberg, *Constant-size commitments to polynomials and their applications*, in Advances in Cryptology - ASIACRYPT 2010, M. Abe, ed., Lecture Notes in Computer Science, Springer, pp. 177–194.

[30] A. Kothapalli, S. Setty, and I. Tzialla, *Nova: Recursive zero-knowledge arguments from folding schemes*, in Advances in Cryptology – CRYPTO 2022, Y. Dodis and T. Shrimpton, eds., Lecture Notes in Computer Science, Springer Nature Switzerland, pp. 359–388.

[31] J. Kwon, *Tendermint: Consensus without mining.* https://tendermint.com/static/docs/tendermint.pdf.

[32] P. Labs, *Developing the most truly decentralized interoperability solution | polymer ZK-IBC.* https://polymerlabs.medium.com/developing-the-most-truly-decentralized-interoperability-solution-polymer-zk-ibc-f0287ea84a2b.

[33] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, *Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings.* https://eprint.iacr.org/2019/099.

[34] Polygon, *Plonky2: Fast recursive arguments with PLONK and FRI.* https://github.com/mir-protocol/plonky2/blob/136cdd053f2175134cddc61abc587f1862e76921/plonky2/plonky2.pdf.

[35] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, *Zerocash: Decentralized anonymous payments from bitcoin*, in 2014 IEEE symposium on security and privacy, IEEE, pp. 459–474.

[36] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, *zkBridge: Trustless cross-chain bridges made practical.* http://arxiv.org/abs/2210.00264.