zkTree: a zk recursion tree with ZKP membership proofs

Sai Deng*and Bo Du†

Polymer Labs

February 16, 2023

Abstract

We present zkTree, a general description of a tree constructed by recursively verifying children's zero-knowledge (zk) proofs (ZKPs) in a parent ZKP node with the ability to fetch a membership proof of user supplied zk proofs. We also describe a construction pipeline such that zkTree can be built and verified on chain with constant gas cost and low data processing pipeline cost. zkTree makes ZKP on-chain verification cost effective by aggregating a large number of user proofs into one root proof. Once the root proof is verified, all user proofs can be verified by providing merkle membership proofs.

zkTree can be implemented using Plonky2 [32], the combination of PLONK [22] and FRI [16], and its root proof is recursively proved in Groth16 [25].

We also demonstrate how to utilize zkTree to verify the default signature scheme of Tendermint [29] consensus by verifying ed25519 signatures [5] in a single proof in the Ethereum Virtual Machine (EVM).

1 Introduction

Zero-knowledge proofs are a powerful tool to protect user privacy and are widely used in blockchains to verify the validity of private transactions, such as Zcash [33]. Another important application of zero-knowledge proofs is compressing computation, where running a short verification on the blockchain can prove that a long computation has been correctly performed off-chain. Verifying the proof of computation can be completed with less time and gas than running the original computation on-chain. zkEVMs [8] [10], zkRollups [26][12] and zkBridges [34] are among this application catalogue.

However, zero-knowledge provers are known for being slow. Typically, the time complexity of a prover is at least linear in the size of the arithmetic circuit. In the real world, the algorithms that are fast on CPUs are not always easily expressed as zk arithmetic circuits. For example, the widely used EdDSA digital signature scheme over curve25519 [5] requires more than 2 million gates in a zk circuit and 12 second proving time [3]. Also the on-chain verification cost is expensive, especially on Ethereum (ETH), the cheapest zk verifier costs around 230k gas [27] and up to 5m gas for a STARK verification [2]. Thus, many innovative applications such as zkBridge and zkIBC [30] cannot be deployed due to long prover proving time and expensive gas costs for on-chain verification.

The main contribution of our work is introducing the zkTree structure and prototyping the zkTree recursive proving pipeline to improve prover time and lower verification costs. By distributing proof generation across different machines and recursively composing proofs through zkTree, the prover has nearly unbounded computation power and fast proving speed. Also, by sharing the same on-chain verifier with zk membership proofs, different systems/companies could share the invariant on-chain verification costs. By saving time and lowering costs, zkTree brings more possibilities for future zk development.

*Email: `sai@polymerlabs.org`; Corresponding author
†Email: `bo@polymerlabs.org`

2 Background

2.1 Zero-Knowledge Proofs

A non-interactive zero-knowledge system includes a prover P and a verifier V. The prover wants to prove that they ran 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 which implements C: $P(x, w) \rightarrow \pi$. The verifier then verifies the proof with $V(x, \pi) \rightarrow$ 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. The on-chain verifier only needs to verify a short proof. Reducing the running time of P and keeping the complexity and cost of V low is the focus of this paper.

There are many zero-knowledge proof protocols [25] [22] [31] [28], and they vary in many properties such as trust minimization, security assumptions, proving time and verification time. Two of the most compelling zero-knowledge technologies in the market today are zk-STARKs [17] and zk-SNARKs [19]. zk-STARK stands for zero-knowledge scalable transparent argument of knowledge, and zk-SNARK stands for zero-knowledge succinct non-interactive argument of knowledge. Zk-SNARKs at their base depend upon elliptic curves for their security, while the base technology for zk-STARKs relies on collision-resistant hash functions. As a result, zk-STARKs doesn't require an initial trusted setup and also achieve quantum-resistance. However, zk-STARKs proof has a far bigger size of the proofs compared to zk-SNARKs, resulting in much longer verification time than zk-SNARKs and costing more gas [21].

2.2 Recursive ZKP

One of the latest advancements in efficient zk proof generation are 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 $P(x_1, \pi_1, x_2, \pi_2, ...) \rightarrow \pi$. The proving circuit implements the constraints of zk proof verifier of the inner proofs. When the verifier verifies the outer proof π, the inner proofs $\pi_1, \pi_2, ...$ are also verified. $V(x, \pi) \rightarrow$ true $\Rightarrow \pi_1, \pi_2, ...$ are true.

The main benefit of recursive proofs is that proof generation can be done in parallel. The total proving work can be divided between multiple computers instead of just one. This yields massive performance gains for proving multiple circuits or for proving a single circuit that can be split into parallelizable parts. The prover does at least linear work in the size of the circuit, so breaking up a larger circuit into smaller circuits will yield a performance gain.

Recursive proof composition was first realized in practice using cycles of elliptic curves [18]. Subsequent works like Halo [20] and Nova [28] continue to improve the recursion speed and verification cost. Plonky2 [32] is the latest implementation based on techniques from PLONK [22] and FRI [16] and only takes 300ms to generate a recursive proof on a 2021 Macbook Air [32]. We use Plonky2 to implement zkTree due to its fast recursion speed.

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 [23] and 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 [29]. Verifying only the Tendermint light client logic on ETH can exceed the block gas limit [13]. With zk technology, the light client computation can be 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 [27]. In this paper, we demonstrate how to utilize zkTree to verify ed25519, the default signature scheme of Tendermint, on-chain in the EVM.

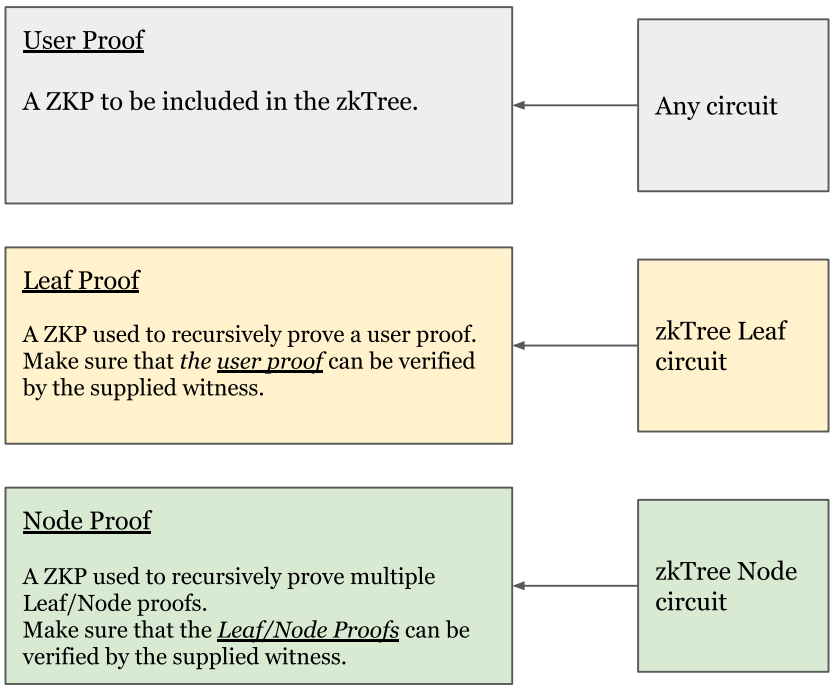

Figure 1: zkTree proof types.

3 zkTree

zkTree is a tree data structure where every node is a zk proof and each parent node recursively proves its children's zk proofs. zkTree is used to prove multiple proofs in a single proof, so the on-chain verification cost can be largely distributed compared to verifying proofs individually. If the root node of zkTree is verified on-chain, then all of the proofs included are also verified on-chain. Figure 2 is an example zkTree proof recursively proving four user proofs.

A non-interactive proof system is a triple (S, P, V). V is the verifier, P is the prover and S is the set of public parameters or common reference strings that usually include two parts: verifier data VD and prover data PD. S is computed by pre-processing the public arithmetic circuits. $\{x\}$ are the public inputs of the proof. For simplicity, private inputs are omitted in this paper.

There are three types of proofs in a zkTree, as shown in figure 1: the user proof π, leaf proof υ and non-leaf node proof ω. User proof π_i may be produced from different circuits with different configurations. Different π_i is associated with different VD_i. Different υ_i is produced by the same Leaf circuit and different ω_i is produced by the same non-leaf Node circuit. Mathematically, a zkTree is constructed by the set of $\{\pi\}$, $\{\upsilon\}$ and $\{\omega\}$ with the following constraints.

User i run

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

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

zkTree Leaf builder run

$$P_l(\pi_i, \{x_i\}, \text{VD}_i) \rightarrow \upsilon_i, h_i, c_i$$

to verify π_i with other inputs and generate the leaf proof υ_i, where $c_i = H(\text{VD}_l || \text{VD}_i)$, 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.

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

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

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

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

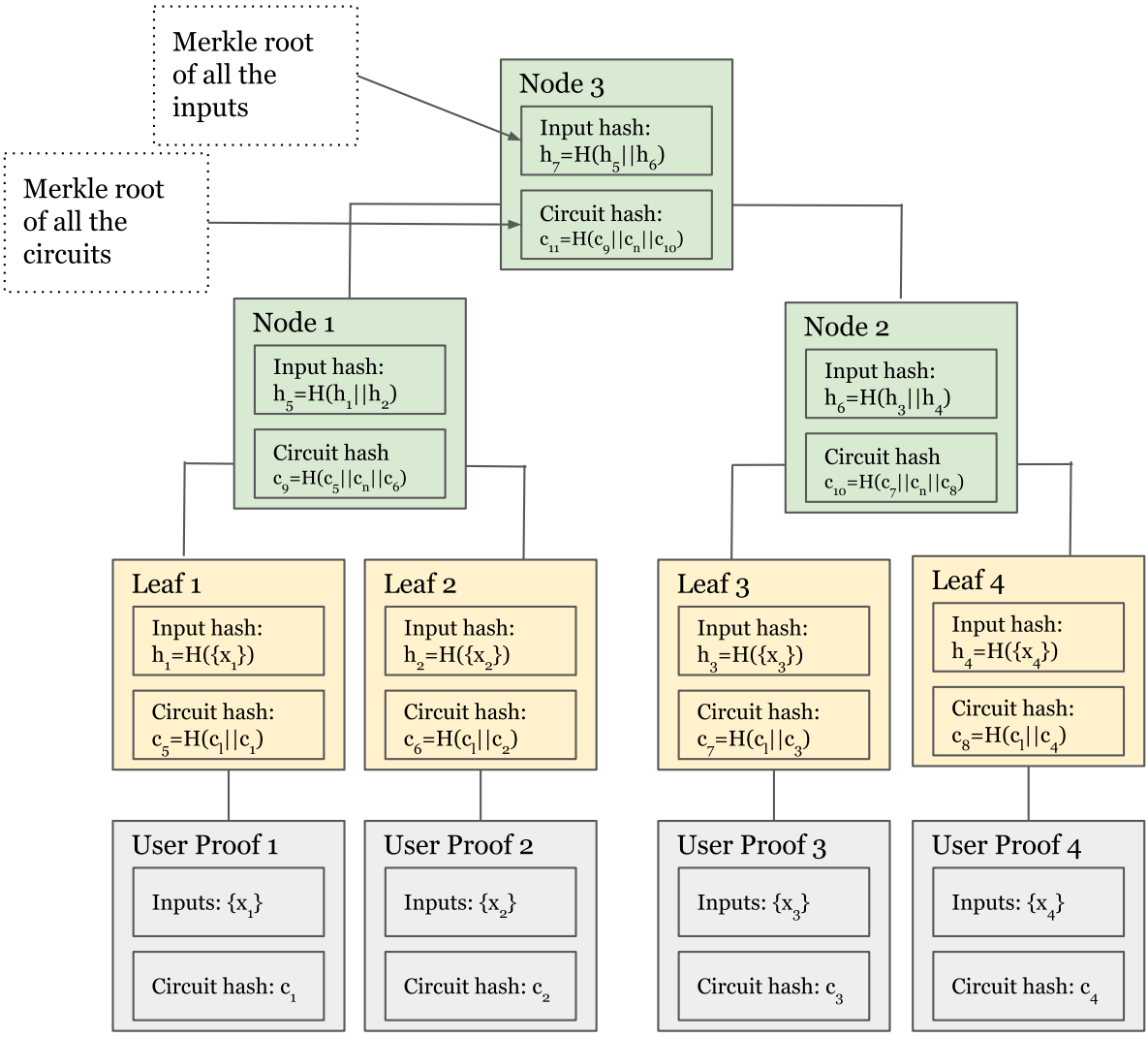

Figure 2: A zkTree example with four user proofs.

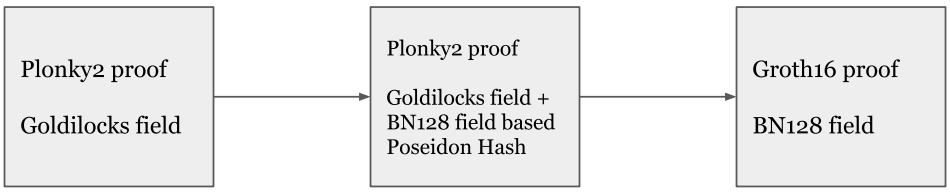

Figure 3: zkTree ETH verifier.

P_n verifies two input proofs in the node circuit. $h_k = H(h_i||h_j)$ and $c_k = H(c_i||\text{VD}_n||c_j)$, 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. The circuit hash and input hash computed in the root node are the merkle roots of all circuit hashes and user proofs. In order to verify if a user proof is included in the root proof, we just need to verify the merkle path of its input hash and user circuit hash. In the example figure 2, in order to verify user proof 4 is included in root proof Node 3, the circuit hash c_4, c_7, c_9 and the input hash h_4, h_3, h_5 need to be provided. c_l and c_n are the circuit hash of the leaf circuit and non-leaf node circuit, they are public parameters and used to verify the zkTree builder circuits are safe.

4 On Chain Verifier

Plonky2 proofs are expensive to verify on-chain in the EVM. There are two steps in the Plonky2 verifier that are expensive: PLONK custom gates constraints evaluation and FRI protocol verification. Both of the two steps require heavy computations such as arithmetic operations over the Goldilocks field [32] and hashing. Even if the hash function is replaced with an EVM-friendly hash function such as keccak256, which has a special EVM op code costing 36 gas cost per operation, the total gas costs are still infeasible. For example, the gas costs of verifying a 50KB Plonky2 proof in the EVM with keccak256 hashing is around 18M gas [11].

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

To recursively prove a ZKP, the complexity is the sum of prover complexity and verifier complexity. For a FRI-based protocol such as Plonky2, hashing is the main bottleneck for both the prover and the verifier [16]. Plonky2 uses Goldilocks based Poseidon [24] hashing and about 75% of the recursive circuit is devoted to hashing operations for verifying merkle proofs [32].

On the other hand, Groth16, PLONK or other KZG based zk-SNARKs schemes [27] requires a pairing-friendly prime field. Since the Goldilocks field is not pairing-friendly, we need to implement the Goldilocks based circuit in a different field such as bn128 or bls12-381. Non-native field operations require expensive range checks and can make up a significant portion of the recursive circuit. To optimize the Groth16 circuit and reduce the overall proving time, we introduce an intermediate Plonky2 proof using a poseidon based hash function over a pairing-friendly field in the Plonky2 prover. The Groth16 circuit size is optimized by getting rid of many non-native range checks when running hashing in the FRI protocol verification. For example in the ETH zkTree verifier in figure 3, an intermediate Plonky2 proof using a bn128 field based poseidon hash function is used as the middle recursive proof.

5 Distributed Proof Generation

Unlike the work in [34] which is based on circuit splitting and proof aggregation, in zkTree the proofs of the same level are independent, there is no communication cost to generate the same level proofs. In theory, zkTree generation time is $log(n)$ multiply the time of proving a node proof, where n is the number of user proofs, and the total communication cost is n^2 multiply the size of a proof. The size of a proof is around 130kb for a Plonky2 based zkTree construction.

Due to the flexibility of zkTree generation, the zkTree ingestion pipeline can be highly customized in production. Based on the hardware configuration of the workers, multiple nodes can be generated in a single worker. As shown in figure 4, worker 1 can have less computation load than worker 2. Communication cost is reduced with heterogenous load distribution. The zkTree ingestion pipeline

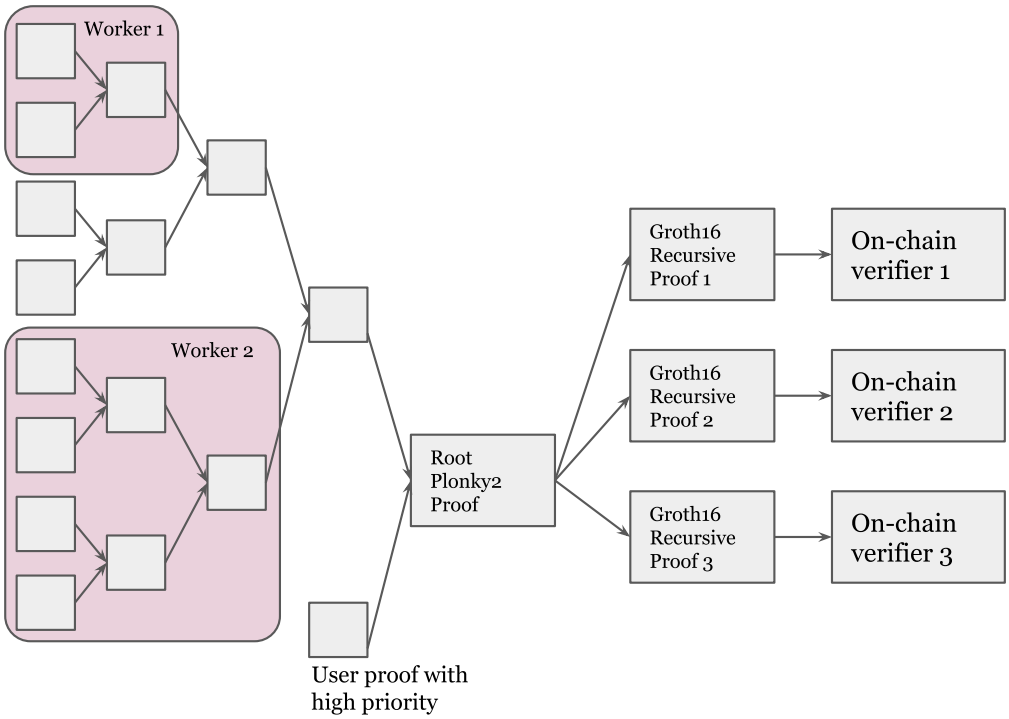

Figure 4: zkTree generation.

could also be implemented with a data streaming engine. When a high priority user proof arrives it could be immediately included in a zkTree construction at a shallower depth than other user proofs.

If further speedup is needed, the Groth16 circuit that is used to verify Plonky2 proof can be split into multiple sub-circuits. For example, each FRI_QUERY_ROUND can be put into a single circuit while the hash of the public inputs and outputs of each sub-circuit must be verified on chain to make sure that they are from the same Plonky2 proof. By splitting the Groth16 we're trading off between prover speed and on-chain verification cost. Verification costs increase with the number of sub-circuits.

6 Results

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

To demonstrate the practicality of zkTree, we implemented a prototype to verify ed25519 signatures and compare the results with the other state-of-the-art proving systems. In a Tendermint based blockchain, using Cosmos as an example, each block header contains about 128 EdDSA signatures (using SHA-512 and Curve25519), where 32 top signatures are required to achieve super-majority stakes [14]. To verify Ed25519 proofs on ETH, we need to simulate curve25519 on curve bn128. This leads to large circuits and long prover time. Thanks to zkTree, we can distribute the Ed25519 proof generation across different machines and use zkTree to aggregate these proofs.

The test machine we use for zkTree proof generation is a Google Cloud n2d-highcpu-64 with 64 vCPUs and 64GB RAM. The machine running Circom witness generation and Rapidsnark prover is a Google Cloud n2-highmem-32 with 32 vCPUs and 256Gb RAM [4].

In Table 1, we list the time needed for each stage of the pipeline. As described in the previous section, when the Groth16 recursive circuit is split, the prover time decreased and the on-chain verification costs increased. The balance between cost and time can be adjusted based on the requirement of the use case and is shown in figure 5. In the ed25519 proof generation stage, the Plonky2 ed25519 circuit can be split into sub-circuits to be included into zkTree to reduce the time. It can be future work or an area of further exploration.

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.

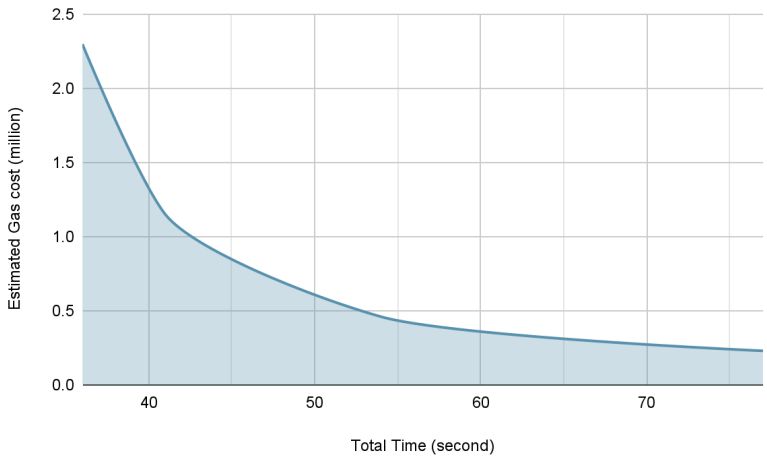

Figure 5: zkTree runtime and cost estimation of verifying 32 ed25519 on ETH.

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 compared the speed and cost of some state-of-the-art systems to generate 32 ed25519 proofs. All of the methods use Groth16 as the final on-chain verification proof. It is worth mentioning that the verification cost of zkTree can be drastically reduced by sharing the cost with other systems by including more proofs in the tree. While deVirgo [34] has the lowest total runtime, it relies on a central machine to communicate between ordinary machines which is more vulnerable to single point of failure. When the deVirgo circuits become massive, for example if 5k proofs need to be aggregated the communication cost would be around 5TB. Network bandwidth and the primary machine's hardware would become bottlenecks. Lastly, since deVirgo is based on circuit-splitting it is not as flexible as zkTree. It requires a re-deploy the updated sub-circuits to every machine when there is any change in the circuit.

7 Conclusion and Future Work

zkTree enables fast and cheap recursive composition of zk proofs. Thousands of ZKPs can be recursively composed and verified on-chain with merkle membership proofs around one minute and 230k gas in one Groth16 proof. zkTree is flexible, and its cost and speed can be re-balanced based on different use case scenarios. The Groth16 proof in the last step can be replaced with a PLONK proof, which does not require per-circuit trusted setup. With custom gates added, we can remove the immediate proof in figure 3 and directly verify the Plonky2 proof in a PLONK proof in less time. With the help of hardware acceleration such as FPGA and ASIC, the Plonky2 and Groth16 prover can be further accelerated [15], so that the total time of 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] *Scroll -a native zkEVM layer 2 solution for ethereum*. https://scroll.io/.

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

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

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

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

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

[16] 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.

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

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

[19] 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.

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

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

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.