Efficient Layered Circuit for Verification of SHA3 Merkle Tree

Changchang Ding^{*} Zheming Fu[†]

Abstract

We present an efficient layered circuit design for SHA3-256 Merkle tree verification, suitable for a GKR proof system, that achieves logarithmic verification and proof size. We demonstrate how to compute the predicate functions for our circuit in $O(\log n)$ time to ensure logarithmic verification and provide GKR benchmarks for our circuit.

1 Introduction

The Zero-Knowledge Proof (ZKP) is a crucial foundational element that supports various essential infrastructures, including remote authentication, electronic voting, cryptocurrency, and decentralized finance. ZKPs can typically be designed to be non-interactive and transparent, addressing even more practical demands. As a potential breakthrough for enhancing security and privacy in cyberspace, substantial research and industrial investment have been directed toward creating faster and more compact Non-Interactive Zero-Knowledge (NIZK) proofs.

Many cutting-edge zero-knowledge proof systems [2,5–7] are based on GKR, a foundational randomized protocol that converts the validity of layered circuit computations into significantly fewer (primarily linear) polynomial constraints. The key differences among these protocols arise mainly from their approaches to proving the resulting polynomial constraints.

To archive fast verification and proof size, the crucial part to prove a particular computation in GKR is too design an efficient layered circuit with logarithmic predict functions for each layer. One should consider whether they need to include more witness in the input to simplify the computation and how the layers should be constructed to provide more pattern for the predicate functions.

An important application for ZKP is the verification of Merkle tree opening on one commitment. And we try to use the standard SHA3-256 for the hash function in the tree construction. In this paper, we will show that it is possible for a layered circuit to achieve fast verification on this kind of application.

^{*}Indiana University, dingchan@iu.edu.

[†]Corresponding author. Post-Doctoral Research Center, China Reinsurance (Group) Corporation, Beijing 100033, China. fuzm@chinare.com.cn.

2 Preliminaries

2.1 GKR

Multilinear Extension. The multilinear extension of any function $V : \{0,1\}^{\ell} \mapsto \mathbb{F}$ is denoted by $\widetilde{V} : \mathbb{F}^{\ell} \mapsto \mathbb{F}$, a polynomial defined as

$$\widetilde{V}(x_0,\ldots,x_{\ell-1}) \stackrel{\text{def}}{=} \sum_{b \in \{0,1\}^\ell} \left(V(b) \prod_{i \in \ell} \left((1-x_i)(1-b_i) + x_i b_i \right) \right).$$

Obviously, $\forall x \in \{0,1\}^{\ell}, \widetilde{V}(x) = V(x)$. Using multilinear extension, the validity of *all* evaluations of V can be reduced to checking a single uniform random point of \widetilde{V} , except for a soundness error of $1/|\mathbb{F}|$.

SumCheck. The goal of the sumcheck protocol [4] is to verify the summation of a polynomial $f : \{0, 1\}^{\ell} \mapsto \mathbb{F}$ on a binary hypercube, i.e., $\sum_{b_i \in \{0,1\}} f(b_1, \ldots, b_{\ell})$. It is achieved by reducing a correct summation into evaluating \tilde{f} on a random point, i.e., $\tilde{f}(r_1, \ldots, r_{\ell})$ with $r_i \in \mathbb{F}$ uniformly picked by \mathcal{V} in each of the ℓ rounds. In the i^{th} round, \mathcal{P} sends polynomial

$$\widetilde{f}_i(x_i) \stackrel{\text{def}}{=} \sum_{b_{i+1},\dots,b_\ell \in \{0,1\}} \widetilde{f}(r_1,\dots,r_{i-1},x_i,b_{i+1},\dots,b_\ell)$$

where r_1, \ldots, r_{i-1} are random values picked by \mathcal{V} in previous rounds. Then \mathcal{V} checks

$$\widetilde{f}_{i-1}(r_{i-1}) = \widetilde{f}_i(0) + \widetilde{f}_i(1)$$

and sends random $r_i \in \mathbb{F}$. Overall, \mathcal{V} checks ℓ linear equalities and the validity of $\tilde{f}(\mathbf{r})$. The i^{th} SumCheck round introduces soundness error $\deg(f_i)/|\mathbb{F}|$.

GKR. Given a *d*-layer circuit, the GKR protocol [3] captures the correctness of the i^{th} layer computation with a multivariate polynomial $V_i : \{0, 1\}^{s_i} \to \mathbb{F}$ defined by

$$V_{i}(z) = \sum_{x,y} \left(\mathsf{add}_{i}(x,y,z) \left(V_{i+1}(x) + V_{i+1}(y) \right) + \mathsf{mul}_{i}(x,y,z) V_{i+1}(x) V_{i+1}(y) \right)$$

where $\mathsf{add}_i, \mathsf{mul}_i$ are the predicate functions for addition and multiplication gates on the i^{th} layer.

The GKR protocol starts from using multilinear extension to reduce the validity of all outputs of the circuit to $\widetilde{V}_0(\mathbf{t}_0)$ where \mathbf{t}_0 is a random point picked by \mathcal{V} . Then, for *i* from 1 to d-1, GKR uses SumCheck to verify $V_{i-1}(\mathbf{t}_{i-1})$, reducing it to checking the values of $\widetilde{V}_i(\mathbf{r}_i), \widetilde{V}_i(\mathbf{s}_i)$ and a equality:

$$v = \widetilde{\mathsf{add}}_i(\boldsymbol{r}_i, \boldsymbol{s}_i, \boldsymbol{t}_{i-1}) \big(\widetilde{V}_i(\boldsymbol{r}_i) + \widetilde{V}_i(\boldsymbol{s}_i) \big) + \widetilde{\mathsf{mul}}_i(\boldsymbol{r}_i, \boldsymbol{s}_i, \boldsymbol{t}_{i-1}) \widetilde{V}_i(\boldsymbol{r}_i) \widetilde{V}_i(\boldsymbol{s}_i)$$

where v is a value \mathcal{P} claimed in the last SumCheck round and $\mathbf{r}_i, \mathbf{s}_i, \mathbf{t}_{i-1}$ are random challenges picked by \mathcal{V} in previous SumCheck rounds. Finally, at the initial input layer, the validity of any evaluation of \widetilde{V}_d can be established by interpolating the multilinear polynomial defined by the circuit's initial inputs. One concern of this scheme is that the number of points to check on \widetilde{V}_i doubles as *i* increments. Chiesa et al. [1] proposed to fix this issue by combining $\widetilde{V}_i(\mathbf{r}_i)$ and $\widetilde{V}_i(\mathbf{s}_i)$ at each layer using public-coin random coefficients.

2.2 SHA3

Keccak is a family of sponge functions that has been standardized in the form of SHAKE128 and SHAKE256 extendable output functions and of SHA3-224 to SHA3-512 hash functions in FIPS 202.

In Keccak, the underlying function is a permutation chosen in a set of seven Keccak-f permutations, denoted Keccak-f[b], where b is the width of the permutation and b = 1600 for SHA3-256. The width of the permutation is also the width of the state in the sponge construction. The state is organized as an array of 5x5 lanes, each of length w = b/25.

Detailed pseudo-code for the permutation and sponge functions are described in Figure 1.

2.3 Merkle Tree

Merkle hash tree proposed by Ralph Merkle in [29] is a common primitive to commit a vector and open it at an index with logarithmic proof size and verification time.

For merkle tree with n leaves, each merkle tree proof consists of the $\log n$ number of sibling nodes of each path nodes and additional $\log n$ number of bits indicating the left or right of each path nodes.

The verification process is described in Figure 2.

3 Circuit Design

3.1 Circuit Layout

The input layer is parallel divided into $\log n$ blocks, each of which has 512 inputs. The first blocks has 128 inputs for the message, 64 inputs for the left-right selectors padding with 64 input of zeros and finally 256 inputs for the root-hash. The rest blocks of the input layer has the same pattern: 256 inputs of the prev-hash and 256 inputs of the sibling hash.

The second layer will decompress the input layer by extending each block into 1024 gates. In the first block, the message input is extent to 512 wires with 384 padding zeros, the rest 512 gates is relaying the 256 root-hash with 256 padding zeros. The rest blocks each has 256 selected left hash, 256 selected

```
\texttt{Keccak-}f[b](A) {
  for i in 0..n-1
     A = \texttt{Round}[b](A, RC[i])
  return A
}
Round[b](A, RC) {
  \# \theta step
  C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], for x in 0..4
  D[x] = C[x-1] \oplus rot(C[x+1], 1), for x in 0.4
  A[x,y] = A[x,y] \oplus D[x], for (x,y) in (0..4, 0..4)
  \# \rho and \pi steps
  B[y, 2 * x + 3 * y] = rot(A[x, y], r[x, y]), for (x, y) in (0..4, 0..4)
  \# \chi step
  A[x,y] = B[x,y] \oplus ((\sim B[x+1,y]) \cdot B[x+2,y]), \quad \text{for } (x,y) \text{ in } (0..4,0..4)
  \# \iota \text{ step}
  A[0,0] = A[0,0] \oplus RC
  return A
}
\operatorname{Keccak}[r, c](\operatorname{Mbytes} || \operatorname{Mbits}) 
  # Padding
  d = 2^{|Mbits|} + sum for i = 0.. |Mbits| - 1 of 2^{i*Mbits[i]}
  P = Mbytes || d || 0x00 || \dots || 0x00
  P = P \text{ xor } (0 \times 00 || \dots || 0 \times 00 || 0 \times 80)
  \# Initialization
  S[x,y] = 0, for (x,y) in (0..4, 0..4)
  # Absorbing phase
  for each block Pi in P
     S[x,y] = S[x,y] xor Pi[x+5*y], for (x,y) such that x+5*y < r/w
     S = Keccak-f[r+c](S)
  # Squeezing phase
  Z = empty string
  while output is requested
     Z \;=\; Z \;\; | \; | \;\; S \left[ \, x \, , \, y \, \right] \,, \;\; \text{for } \;\; ( \, x \, , \, y \, ) \;\; \text{such that } \; x + 5 * y \; < \; r \, / w
     S = Keccak-f[r+c](S)
  return Z
}
```

Figure 1: SHA3-256 Pseudo-Code

```
Verify(Root, Open, Siblings, Left-Right) {
    k = Siblings.length
    m = SHA3-256(Open)
    for i in 0..k - 1
        if Left-Right[i] == 0
            m = SHA3-256(m || Siblings[i])
        else
            m = SHA3-256(Siblings[i] || m)
        return m == Root
}
```

Figure 2: SHA3-256 Merkle Tree Verification

right hash, 256 bit prev-hash and 256 bit sibling hash. After this arrangement, the first 512 wires of each block is the pre-image to each SHA3-256.

The next 48 layers corresponds the 24-iteration loop of SHA3-256 round, each consists of one linear layer for the θ , ρ , π and ι steps, and one non-linear layer for the χ step. The width of each block is further extent to 4096, where the first 1600 gates are used to store the SHA3-256 inner states. The rest gates are storing the pre-hash which should be the result of the SHA3-256 in-block and 256 bit-tests for sibling hash.

In the output layer, we will test the SHA3-256 output with the prev-hash by outputting the subtraction results, and also output the bittest results. Therefore, a correct computation will output zeros on all the output gates.

3.2 Predicate Functions

3.2.1 In-block predicates

The predicate function for a kind of gate that has a pattern that all of the input and output wires are in the same block can be easily logarithmic paralleled. For example, a fan-in-2 gates g in a m-bit layer with k-bit block has

 $pred_a(x, y, o) =$

$$eq(x_k..x_{m-1}, y_k..y_{m-1}, z_k..z_{m-1}) \cdot sub-pred_g(x_0..x_{k-1}, y_0..y_{k-1}, z_0..z_{k-1}).$$

where

$$eq(x_k..x_{m-1}, y_k..y_{m-1}, z_k..z_{m-1}) = \prod_{i=k}^{m-1} (x_i y_i z_i + (1-x_i)(1-y_i)(1-z_i))$$

It can be easily seen that the cost is $O(m - k) = O(\log n)$, where n is the number of blocks.

3.2.2 Inter-block predicates

Now we consider a more complicated case where at least one of the input wire is in another block. In our case, since each SHA3-256 will use the SHA3 result from the previous hash, i.e. the previous block, for a fan-in-2 g gate with input x, y, z, we can assume that the first input x is from the previous layer while yand z are in the same block.

 $\operatorname{pred}_{q}(x, y, z) =$

 $inter-block-pred(x_k..x_{m-1}, y_k..y_{m-1}, z_k..z_{m-1}) \cdot sub-pred_q(x_0..x_{k-1}, y_0..y_{k-1}, z_0..z_{k-1}).$

The idea is to test the least significant bit of x, if it is zero, then the least significant bit of y is one and the rests are the same with x. Otherwise, the least significant bit of y should be zero and we will recursively call the previous by removing the least significant bit of x and y. And we should also consider the output z, which is always the same as y.

inter-block-pred $(x_k..x_{m-1}, y_k..y_{m-1}, z_k..z_{m-1}) =$

$$(x_k == 0 \land y_k == 1 \land z_k == 1) \cdot eq(x_{k+1}..x_{m-1}, y_{k+1}..y_{m-1}, z_{k+1}..z_{m-1}) + (x_k == 1 \land y_k == 0 \land z_k == 0) \cdot inter-block-pred(x_{k+1}..x_{m-1}, y_{k+1}..y_{m-1}, z_{k+1}..z_{m-1})$$

where

$$(x_k == 0 \land y_k == 1 \land z_k == 1) = (1 - x_k)y_k z_k$$

$$(x_k == 1 \land y_k == 0 \land z_k == 0) = x_k(1 - y_k)(1 - z_k)$$

Since the number of recursion is $m - k = \log n$, the cost is still $O(\log n)$.

4 Experiment

We experiment our SHA3-256 merkle tree verification circuit with GKR protocol with Rust code. The binary field we have tested is 192-bit which can easily achieve 128-bit security for the GKR protocol. All the results are obtained on a Linux laptop with Intel i9-11900H CPU and 64GB DDR4.

The results in Table 1 clearly shows a logarithmic verifier and proof size.

5 Conclusion

We propose a new design of layered circuit for SHA3-256 Merkle tree verification, with logarithmic predicate functions for each layer, which offers logarithmic time for GKR verification.

$\log n$	Prover time (ms)	Verifier time (ms)	Proof size (kB)
2	83.70	24.283	48.93
3	163.63	24.534	52.49
4	321.09	25.196	56.05
5	654.46	25.491	59.62
6	1301.87	25.991	63.18
7	2577.37	27.306	66.74

Table 1: Experiment Result

References

- [1] Alessandro Chiesa, Michael A Forbes, and Nicholas Spooner. A zero knowledge sumcheck and its applications. arXiv:1704.02086, 2017.
- [2] Changchang Ding and Yan Huang. Dubhe: Succinct zero-knowledge proofs for standard aes and related applications. In USENIX Security, 2023.
- [3] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC*, 2008.
- [4] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.
- [5] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Symposium on Security and Privacy*, 2018.
- [6] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, 2019.
- [7] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE Symposium on Security and Privacy*, 2020.