Benchmarking circuits of Keccak256 hash function using Zero-Knowledge protocols.

July 2023

Abstract

This document compares three implementations (Maru, Axiom, and JumpCrypto) for circuit building, proof generation, proof size, and proof verification in Keccak-256. When comparing the Maru implementation with the JumpCrypto and Axiom implementations for different message sizes, the Maru implementation outperforms both in terms of speed. Specifically, for messages up to 20KB, the Maru implementation is faster than the JumpCrypto implementation. Similarly, for messages up to 50KB, the Maru implementation is faster than the Axiom implementation. JumpCrypto implementation, however, suffers from slower circuit building and has a dependency on the number of blocks for the input message length, offers faster proof verification. Axiom has the lowest proof size and provides consistent proof generation time for larger messages. In terms of effectiveness compared to native Keccak verification, JumpCrypto performs better for smaller message lengths, while Axiom and Maru intersect with native verification at larger message lengths. The comparison of different implementations' trend lines showed that JumpCrypto's implementation is the most efficient for message lengths around one million bytes compared to native Keccak verification. Axiom's implementation becomes more efficient at around 24 million bytes. Maru's implementation is also like Axiom's and intersects with the native verification trend line at approximately 24 million bytes. Ultimately, the choice of implementation depends on specific use cases and priorities.

1 Introduction

One of the technologies that is currently gaining popularity and will transform not only cryptography, but also improve all existing blockchain infrastructures in the future is Zero-knowledge cryptography, which offers advantages in confidentiality, security, and data integrity. Keccak function is based on sponge function. Sponge function basically provides a particular way to generalize hash functions. It is a function whose input is a variable sized length string and output is a variable length based on fixed length permutation. Sponge construction consists of some of the terms. The operations taking place within the permutation blocks. In this case, the function employed is f = Keccak-f[1600], which was tested in this paper. It utilizes a permutation that incorporates AND, NOT, and XOR operations. The state represents an array consisting of bits arranged in a 5x5xw format, where w is defined as w=power(2,l). Since we have chosen l=6, w is equal to 1600. Therefore, we utilize 1600 bits as the input length S. We denote the bit (5i + j) × w + k of the input as a[i][j][k]. The block permutation function involves twelve plus two times l rounds, each comprising of five steps: $\theta(theta)$, $\rho(rho)$, $\pi(pi)$, $\chi(chi)$, and $\iota(iota)$. This hash function is used in many cryptographic systems, but its main advantage lies in its usage in blockchain networks. For example, Ethereum utilizes it to build Merkle trees efficiently, which store and verify the integrity of large amounts of data, such as transaction histories and account balances. Transitioning the blockchain to zero-knowledge requires a significant amount of time to improve these protocols. The current efficiency is slower than native verification. This paper describes testing different circuit implementations of Keccak, utilizing various frameworks to build proofs of correctness for the Keccak. The main idea of the testing is to demonstrate the effectiveness of different approaches using various protocols and analyze the efficiency of these proofs using zero-knowledge frameworks such as:

- Halo2 (KZG) - `https://github.com/axiom-crypto/halo2-lib`: Halo2-lib is Axiom's implementation of zk-SNARK with Plonk. It utilizes a PLONKish arithmetization that includes support for custom gates and lookup tables.
- Plonky2 - `https://github.com/mir-protocol/plonky2`: Plonky2 is a SNARK implementation based on techniques from PLONK and FRI from Polygon Zero. Plonky2 uses a small Goldilocks field and supports efficient recursion.
- Starky - `https://github.com/mir-protocol/plonky2/tree/main/starky`: Starky is a highly performant STARK framework from Polygon Zero.

2 Related implementations

We came across a limited number of unofficial GitHub projects related to the performance of Zero-Knowledge Keccak circuits, as practical implementations in this area are still in their early stages:

- Implementation by Maru. They provide recursive Keccak circuit using plonky2 and starky. The implementation build proof in STARK aggregated to SNARK.
- Implementation by Axiom (`https://github.com/axiom-crypto/halo2-lib/tree/community-edition/hashes/zkevm-keccak/src`). They provide SNARK keccak circuit using Halo2.

- Implementation by JumpCrypto (`https://github.com/JumpCrypto/plonky2-crypto`). They provide SNARK Keccak circuit using plonky2.

The testing objectives focus on validating the computational integrity of the Keccak hash function using circuits. The paper has made significant contributions, particularly through benchmarking efforts. These benchmarks compare the performance of circuits implemented in different frameworks and using various Zero-Knowledge (ZK) protocols. The aim of this contribution is to provide the community to access the performance and future potential of ZK technology. As more circuits and frameworks emerge, there will be additional opportunities for comparison. The benchmarks are readily accessible on GitHub, and anyone interested can easily contribute by following the provided link: `https://github.com/proxima-one/keccak-circuit-benchmarks`. For a comprehensive understanding and testing of these circuits, refer to the link provided above.

3 Benchmark Methodology

To benchmark these various proving systems, we computed the Keccak hash for N bytes of data, where we experimented with N = 136, 272, . . . , 136KB (with one exception being JumpCrypto, where needed more resources. For this implementation range is N = 136, 272, . . . , 2449B). Furthermore, we conducted the benchmarking of each system using the following performance metrics:

- Circuit building time
- Proof generation time
- Proof size in bytes
- Proof verification time

The implementations and their respective parameter ranges are as follows:

- Maru implementation with input message length ranging from 136 to 136,000 bytes.
- Axiom implementation with input message length ranging from 136 to 136,000 bytes. The parameters used are: k=14 and rows=25. This implementation will have 24 possible rounds in the circuit.
- JumpCrypto implementation with input message length ranging from 136 to 2,448 bytes. Larger message sizes do not allow for accurate measurements.

We conducted our benchmarking on Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (4 cores and 8 threads) with 8 GB RAM, but we did not utilize multithreading. The specific version details of OS are as follows:

Linux Version: Ubuntu 22.04.2 LTS

4 Benchmarking

We will compare three implementations by plotting diagrams that include all three implementations.

Circuit building. The fastest circuit building is achieved by the Maru implementation, where the time is constant and takes milliseconds. However, this circuit was built for aggregation, where two proofs (sponge function and permutations) are provided as input, hence the construction process is fast. Next in terms of speed is the Axiom implementation, where the average building time is approximately 2.2 seconds. JumpCrypto implementation shows lower performance due to its dependence on the number of blocks for the input message length. As a result, the circuit building time is very high. In summary, the Maru implementation demonstrates the fastest circuit building, followed by the Axiom implementation, while the JumpCrypto implementation has a relatively slower performance due to its dependency on the number of blocks for the input message length.

Proof generation. Axiom implementation shows the best results for the range of message length. However, considering that blockchain hashes, particularly from Ethereum, have sizes up to 10KB, the Maru implementation demonstrates better performance. If you prioritize larger message lengths in bytes, then the Axiom implementation can be suitable for your needs. It's important to consider your specific requirements. Furthermore, regardless of the message size, the Axiom implementation exhibits a stable proof generation time that increases slowly compared to the other implementations. The JumpCrypto implementation has a similar time for building circuit and proof generation, indicating that both times will increase depending on the message length. The choice of implementation depends on your specific use case and requirements. The Maru implementation performs well for blockchain hashes within the 10KB range, while the Axiom implementation is suitable for larger message lengths.

Proof size. For the given message length range, the Axiom implementation has the lowest proof size, which remains constant unlike the JumpCrypto implementation where the proof size increases for such a small range of message length. The proof size for the Maru implementation is larger but still shows good results. Before aggregation in the STARK scheme, the proof size for permutations was more than 1,780,000 bytes, so the proof size achieved by the Maru implementation is quite impressive. Therefore, if minimizing proof size is a priority, the Axiom implementation would be a favorable choice. However, considering the overall performance and the specific needs of your application, both the Maru and Axiom implementations offer good results in terms of proof size for the given message length range.

Proof verification. Unlike previous comparisons, the advantage of the JumpCrypto implementation lies in the faster verification time, which is significantly faster compared to the other implementations, averaging around 0.0035 seconds. Next in terms of verification efficiency is the Axiom implementation, which has a constant verification time, averaging around 0.08 seconds. The Maru implementation has a higher verification time, averaging around 0.09 seconds for the given range. Although there is a difference in verification time among the implementations, it may not be noticeable when considering the previous comparisons. Therefore, the choice of implementation for verification time may depend on the specific requirements and constraints of your application.

Implementations effectiveness comparing with native verification. The next comparison involves determining the specific point of intersection between the trend lines of each implementation to identify advantage compared to native Keccak verification for message length. By plotting a linear trend line for each range, these trend lines can be extended by determining the slope and intercept. This was done in Python, along with the plots. When the trend lines intersect, it indicates the effectiveness of ZK implementations compared to native verification. The first intersecting trend line belongs to the implementation by JumpCrypto. The proof verification for this implementation occurs faster than others. Therefore, for a size of approximately one million bytes, the verification will be more efficient for this implementation compared to native verification. Furthermore, the trend lines of native verification and proof verification by Axiom will intersect, and they intersect at a size of approximately 24 million bytes. Based on the implementation from Maru, effectiveness comparing to native keccak is precisely the same as Axiom implementation and trend lines will intersect at a size of approximately 24,000,000 bytes.

5 Conclusion

Every day, we work on improving protocols and their frameworks. Although we are still in the early stages of development in this industry, I believe there is a promising future ahead. These tests were conducted to demonstrate different approaches and how effective they are. We are also developing and improving new frameworks that will move us closer to the efficiency of Zero-Knowledge rather than native verification. We evaluated Keccak-256 implementations using three different frameworks and collected metrics on prover time, proof size, verifier time, and specific metrics for each circuit. By comparing these implementations, we aimed to find the point where efficiency favors Zero-Knowledge, but this is just an assumption. Ultimately, the choice of using these implementations depends on your specific use cases and the criteria that matter most to you.

6 Appendix

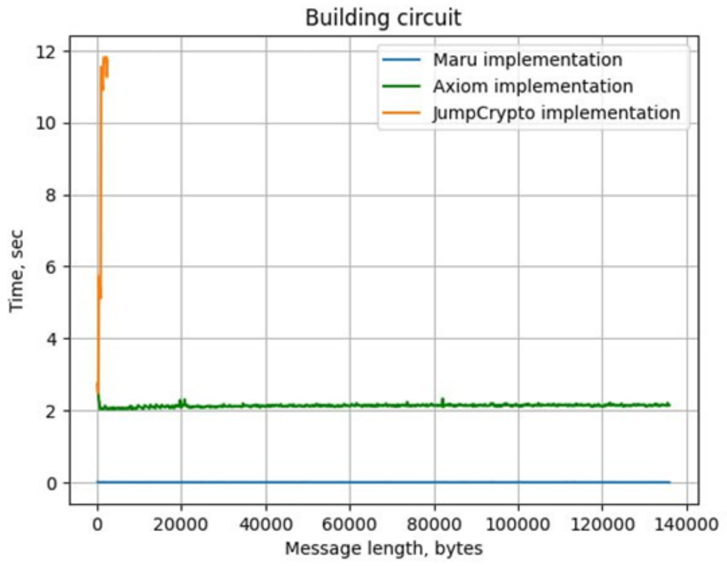

Figure 1: Comparing building circuit time

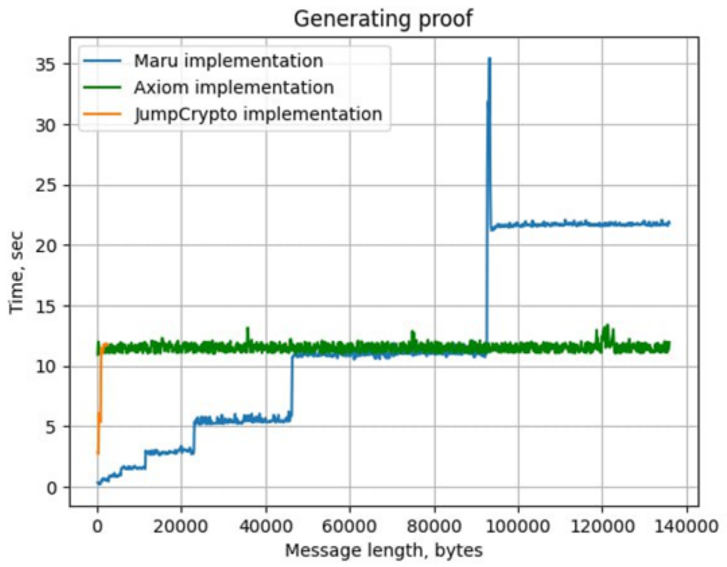

Figure 2: Comparing proof generation time

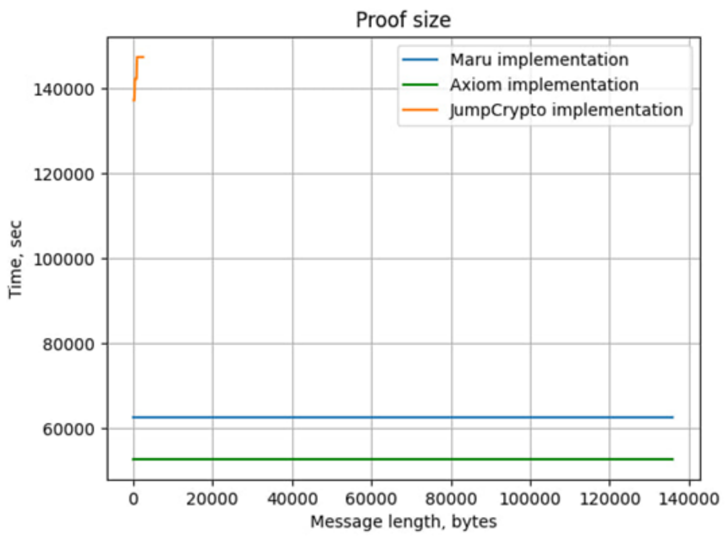

Figure 3: Comparing proof size

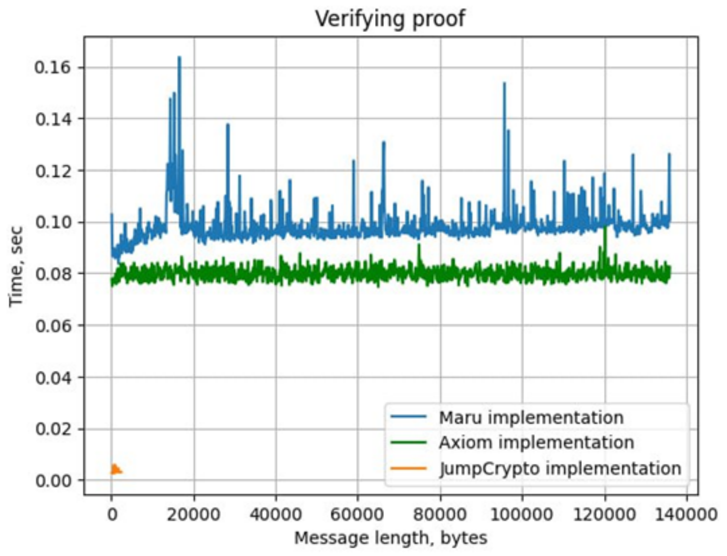

Figure 4: Comparing proof verification time

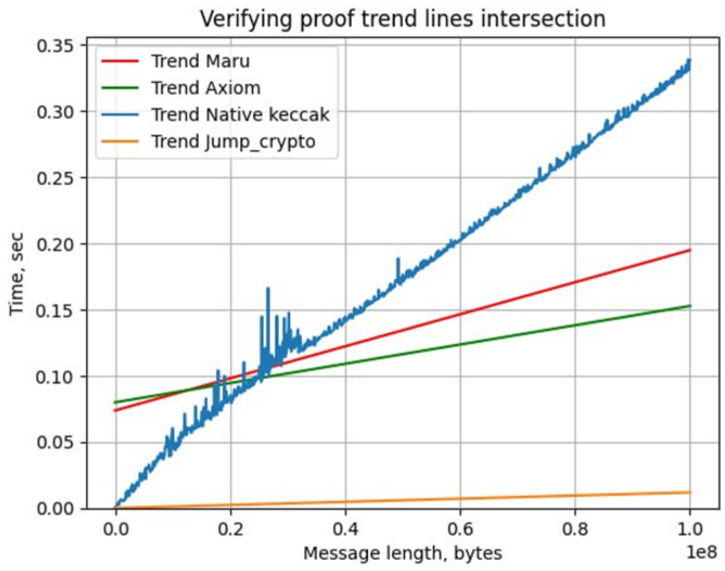

Figure 5: Comparing effectiveness of native keccak with implementations