Benchmarking Keccak256 proving algorithms: Maru, Axiom and JumpCrypto

July 2023

Abstract

Cryptography is constantly evolving, and with it the consensus algorithms that can be used to keep data private. Now, SNARK and STARK are the main technologies that become the most popular in the study. Each of these algorithms has its own pros&cons, as well as ways to use them. This paper compares three proof generation implementations for the Keccak256 hash function from Maru, Axiom, and JumpCrypto that implement SNARK/STARK. We, Maru, offer an implementation of a Keccak256 on STARK, which later "turns" into a SNARK. We also provide the results of a comparison of all three implementations according to such criteria as time for building a circuit, generating & verifying a proof and proof size.

1 Introduction

Zero-knowledge proof is 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. So, we have two participants in the protocol: the prover and the verifier. The prover claims to have certain information. And the verifier must check whether the prover is right. The ZK-SNARK protocol does not require continuous communication or conversation between the prover and verifier. It works based on complex mathematical operations.

ZK-SNARK. The protocol is based on a trusted configuration using such mathematical assumptions as homomorphic functions, blind evaluation, Pinocchio's protoco etc. ZK-SNARK allows users to send transactions on the blockchain in a completely encrypted way. This means that transactions are completely legitimately, but no one can read them. Furthermore, such a transaction cannot be modified by a third party. With ZK-SNARK, we can see that the sender has the funds, but we cannot see how much or where he wants to send it.

ZK-STARK. ZK-STARK provides the ability to share verified data or perform calculations with a third party, without revealing the data to that party. At the same time, it is publicly verifiable. ZK-STARK allows you to verify the banking information of your future business counterparty, without having to disclose your confidential information. The protocol moves computation and storage off the blockchain. Therefore, it improves its scalability and privacy. Services performed outside the blockchain can generate STARK proofs, which simultaneously certify the integrity of off-chain computations. The proofs thus made are then placed back on the blockchain so that any interested party can validate the computation made. This type of zero proof focuses first on scalability and only later on privacy.

SHA3 or Keccak is a hash function consisting of four cryptographic hash functions and two extendable-output functions. These six functions are based on an approach called sponge functions. Sponge functions provide a way to generalize cryptographic hash functions to more general functions with arbitrary-length outputs.

SHA-3 consists of four functional blocks called state function, round constant, buffer function and Keccack function.

The last one 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 $\mathbf{(5i + j) \times w + k}$ of the input as $\mathbf{a[i][j][k]}$. 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 uses it to effectively build Merkle trees, which store and verify the integrity of large amounts of data. Transitioning the blockchain to zero-knowledge requires a significant amount of time to improve these protocols.

This paper describes testing different implementations of Keccak256. The main idea of the testing is to demonstrate different approaches and analyze their efficiency. The implementations we estimate are:

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 proofs, as practical implementations in this area are still in their early stages:

Implementation by Axiom - `https://github.com/axiom-crypto/halo2-lib/tree/community-edition/hashes/zkevm-keccak/src`. They provide SNARK keccak circuit using Halo2.

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

Implementation by Maru. They provide recursive Keccak circuit using plonky2 and starky. The implementation build proof in STARK aggregated to SNARK.

The testing objectives focus on validating the computational integrity of the Keccak hash function using proofs. The benchmarks allow comparing the performance of proofs implemented in different frameworks. 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

In our benchmark we compute the Keccak hash for different data length: 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 verification time
- Proof size in bytes

The implementations and their respective parameter ranges are as follows:

- 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 19,992 bytes. Larger message sizes do not allow for accurate measurements.
- Maru implementation with input message length ranging from 136 to 136,000 bytes.

We conducted our benchmarking on Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (4 cores and 8 threads) with 8 GB RAM (without multithreading). The specific version details: Linux Version: Ubuntu 22.04.2 LTS.

4 Benchmarking

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

Circuit building. In order plot of circuit building to be logical, we scaled the data using logarithmic function, as the JumpCrypto implementation has large values of circuit building time. 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. The JumpCrypto implementation shows slower performance due to its dependence on the number of blocks for the input message length. The maximum time exceeds for 18088 bytes in just 220 seconds, while the minimum time is 4.6 seconds. As a result, the circuit building time is very high. Including, 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. The JumpCrypto implementation show better perfomance than Axiom with message input size up to 2448 bytes. However, the time increases with number of blocks faster than other implementations. 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 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 range of message length, the Axiom implementation has the lowest proof size. 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. After aggregation to SNARK, it was reduced to 62,700 bytes. 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 0.0051 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 check. 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 28 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 32,000,000 bytes.

5 Conclusion

The work demonstrates different approaches and shows their advantages. We evaluated Keccak-256 implementations using three different platforms and collected metrics for verification time, proof size, verification time, and specific metrics for each scheme.

By comparing these implementations, we aimed to find the point at which efficiency favors Zero-knowledge, but this is just a guess. Ultimately, the choice to use these implementations depends on your specific use cases and the criteria that are most important to you.

The crypto community continues to research in search of new frameworks that will bring us closer to efficient data processing. In our opinion, Zero-knowledge proof is a promising direction, the study of which should be continued.

6 Appendix

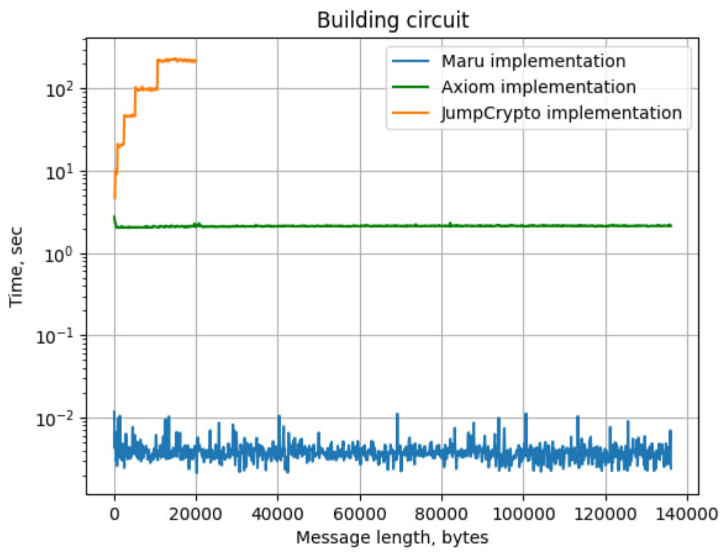

Figure 1: Comparing building circuit time using logarithmic scale function

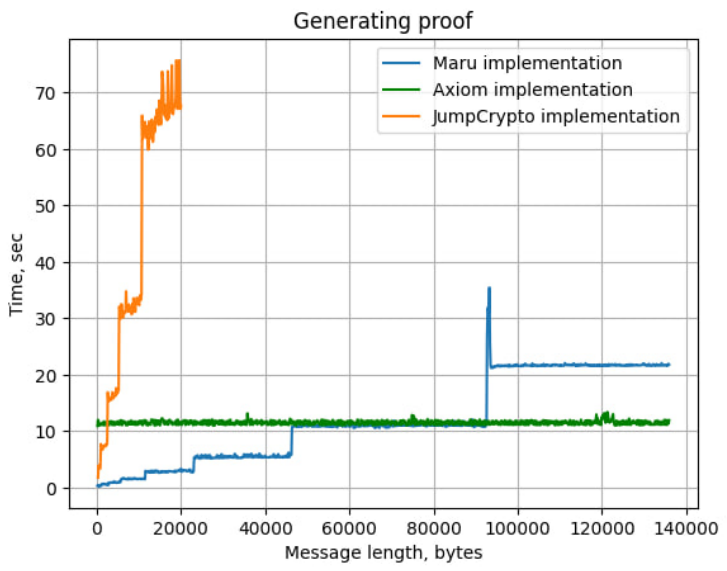

Figure 2: Comparing proof generation time

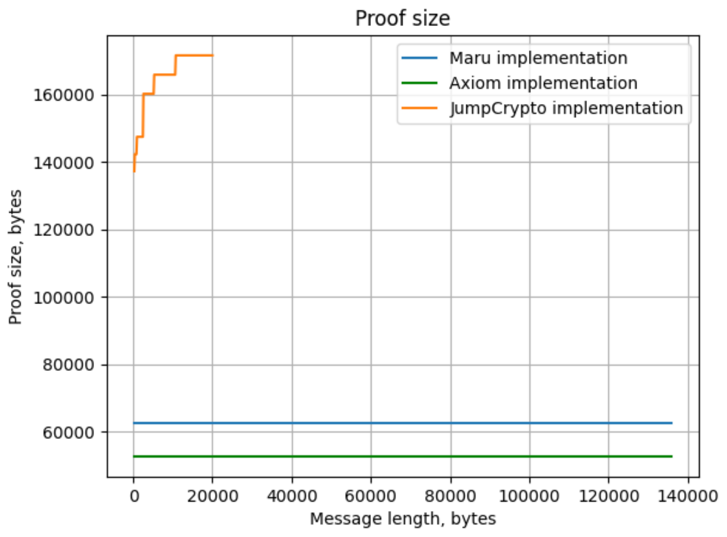

Figure 3: Comparing proof size

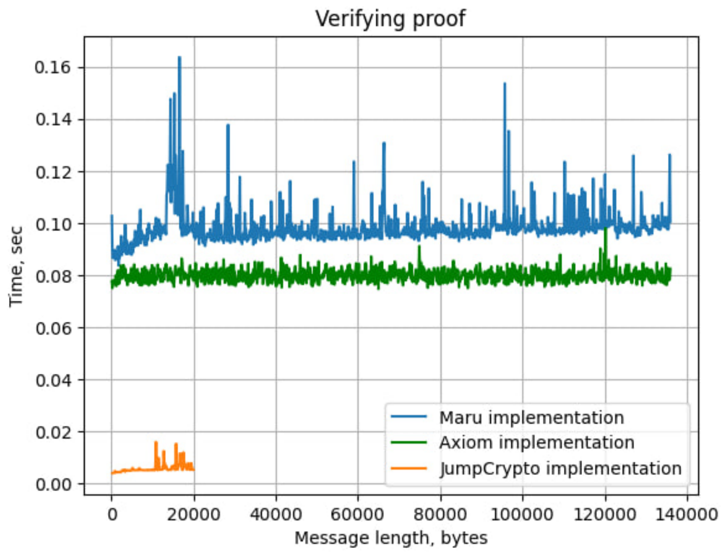

Figure 4: Comparing proof verification time

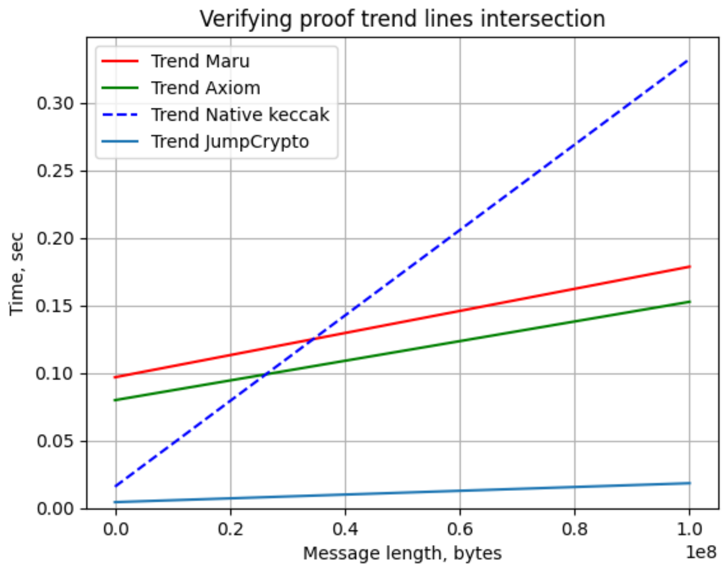

Figure 5: Comparing effectiveness of native keccak with implementations