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. For the time being SNARK and STARK are the main technologies that become the most popular to research. Each of these algorithms has its own pros&cons, as well as the 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 schemes.

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 does not only transform cryptography, but also improves all existing blockchain infrastructures.

Zero-knowledge proof technology allows proving some information without revealing the information itself. The scheme has two participants in the protocol: the prover and the verifier. The prover claims to have certain information. And the verifier has to check the validity of prover's data. 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 protocol 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 Keccak function.

The algorithm receives as a matrix, an input called state, which 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 **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 paper describes testing of different implementations of Keccak256. The main idea of the work is to demonstrate different approaches and analyze their efficiency.

2 Related implementations

We came across a number of GitHub projects, which implements Keccak256 ZK-proofs:

- Axiom provides SNARK Keccak256 proof using Halo2.
  URL: `https://github.com/axiom-crypto/halo2-lib/tree/community-edition/hashes/zkevm-keccak/src`.
- JumpCrypto offers SNARK Keccak256 proof using plonky2. URL: `https://github.com/JumpCrypto/plonky2-crypto`.
- Maru provides recursive Keccak256 circuit using starky and plonky2 frameworks. The idea is to generate a proof in STARK, that turns to SNARK.

Maru implementation of Keccak256 uses Polygon's implementation of Keccak256 for the EVM. In this work, Keccak256 is divided into two entities: sponge and permutation. To solve the problem of computing the computational integrity of a message, we added a check of the public input, i.e. the hash. After

receiving two proofs on STARK, we combine them into one SNARK proof. Such aggregation also allows you to significantly compress the resulting proof.

The benchmark is available on GitHub and anyone can easily contribute by visiting the following link: `https://github.com/proxima-one/keccak-circuit-benchmarks` to understand and test these schemes comprehensively.

Our goal is to show the community the performance and future potential of ZK-technology. As new schemes and frameworks become available, there will be additional opportunities for comparison.

3 Benchmark methodology

In our benchmark we compute the Keccak256 hash function for different data length: N = 136, 272, ..., 136KB (except JumpCrypto, because we need more resources there. We take N = 136, 272, ..., 19992B). We conduct the benchmark of each system using the following performance metrics:

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

The chosen parameters are as follows:

- Axiom implementation with input message length ranging from 136 to 136,000 bytes. The parameters used are: k=15 and rows=25. This implementation will have 50 possible rounds in the circuit.
- JumpCrypto implementation with input message length ranging from 136 to 19,992 bytes. Large message sizes do not allow accurate measurements.
- Maru implementation with input message length ranging from 136 to 136,000 bytes.

We conducted our benchmark on Xeon E5-2697v4, 128GB DDR4 2400MHz with Linux. The specific version details: Linux Version: Ubuntu 22.04.2 LTS.

4 Benchmark

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

Circuit building. As the JumpCrypto implementation has large circuit building time, we scaled the data with a logarithmic function to make the circuit building graph logical. The fastest circuit building is achieved by Maru with a message length less than 48,000 bytes compared to the other two schemes. Note, circuit building time for Maru includes time for calculation of traces for STARK

and circuit building time for SNARK. The next is Axiom, where the average building time is approximately 1.38 seconds. The JumpCrypto implementation shows the slowest performance due to the dependence on the number of blocks of the length of the input message. The maximum time exceeds 18088 bytes in just 220 seconds and the minimum time is 4.6 seconds.

Proof generation. Axiom implementation provides almost constant proof generation time for all the range of message length. However, considering that blockchain hashes, particularly from Ethereum, have sizes up to 10KB, the Maru implementation demonstrates better performance on message sizes of less than 94,000 bytes. If you prioritize larger message lengths, then the Axiom implementation is suitable for your needs. The JumpCrypto implementation shows better performance than Axiom with message input size up to 2448 bytes. But the time for JumpCrypto increases with number of blocks faster than other implementations.

The choice of implementation depends on your specific use case and requirements. The Maru implementation performs well for blockchain hashes within the 94KB range, while the Axiom proposes a better option for larger message lengths.

Proof size. For a given message length range, the Axiom implementation has the smallest proof size. As for Maru, before aggregation, the proof size for permutations is more than 1,780,000 bytes. But we compress it by turning to SNARK and the final size is $\approx$ 63 KB. 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 Maru and Axiom 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 0.125 seconds. The Maru implementation has a higher verification time, averaging around 0.15 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 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 check. By plotting a linear trend line for each range, these trend lines can be extended by determining the slope and intercept. 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 intersect at a size of approximately 37 million bytes. As far as Maru is concerned, the efficiency compared to native checking is the same as the Axiom implementation, and the trendlines will cross at around 48,000,000 bytes.

5 Conclusion

The work demonstrates different approaches and shows their advantages. We conducted the tests of Keccak256 implementations using three different platforms and collected metrics for proof generation and verification time, proof size 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

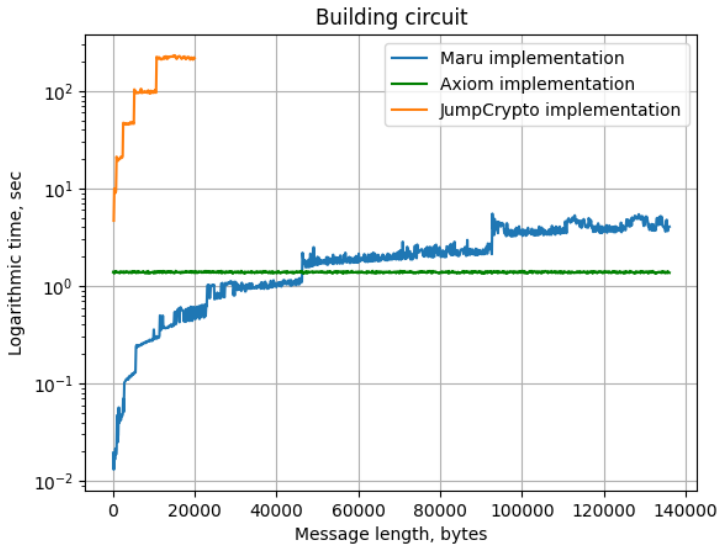

Figure 1: Comparing building circuit time using logarithmic scale function

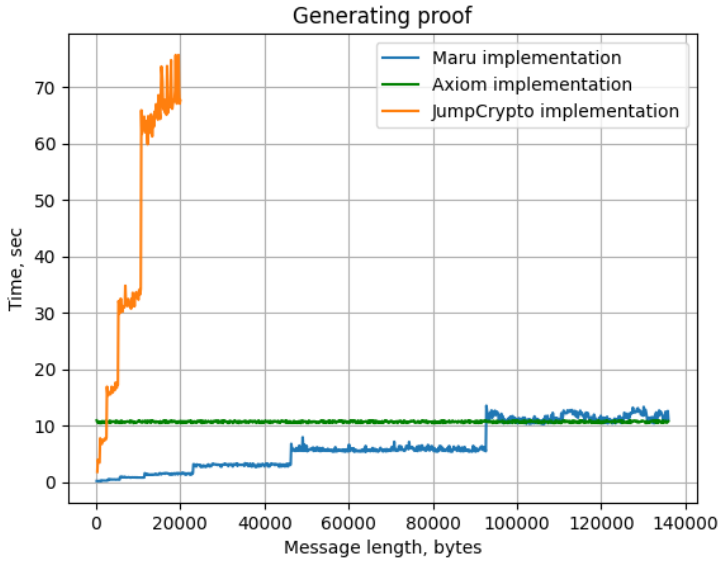

Figure 2: Comparing proof generation time

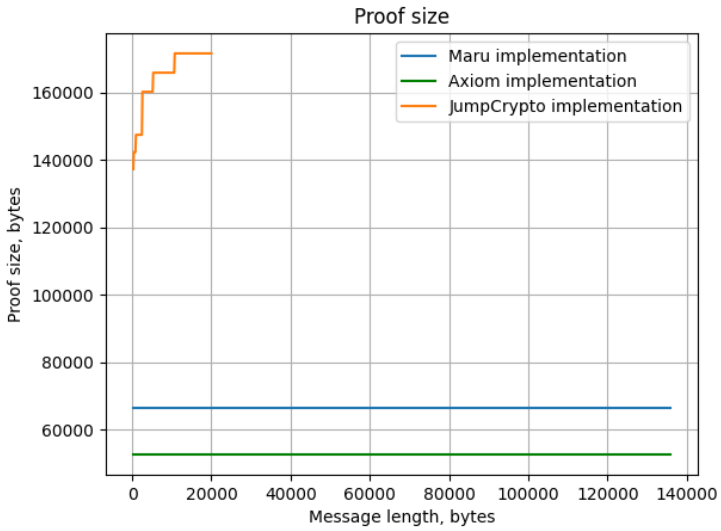

Figure 3: Comparing proof size

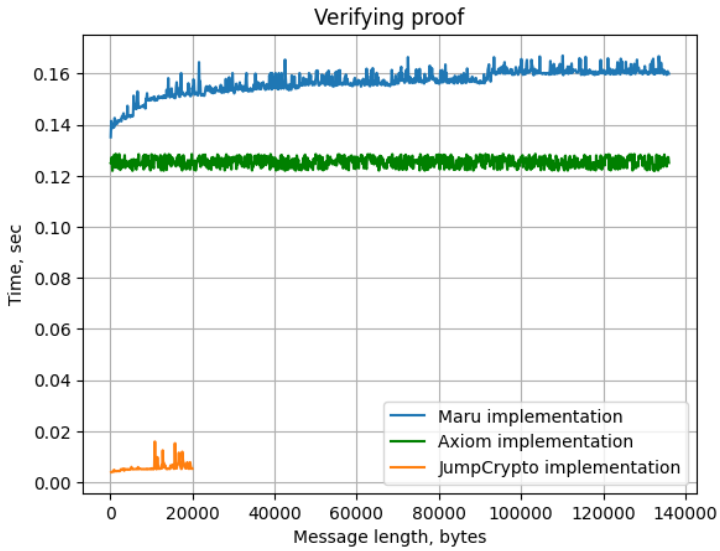

Figure 4: Comparing proof verification time

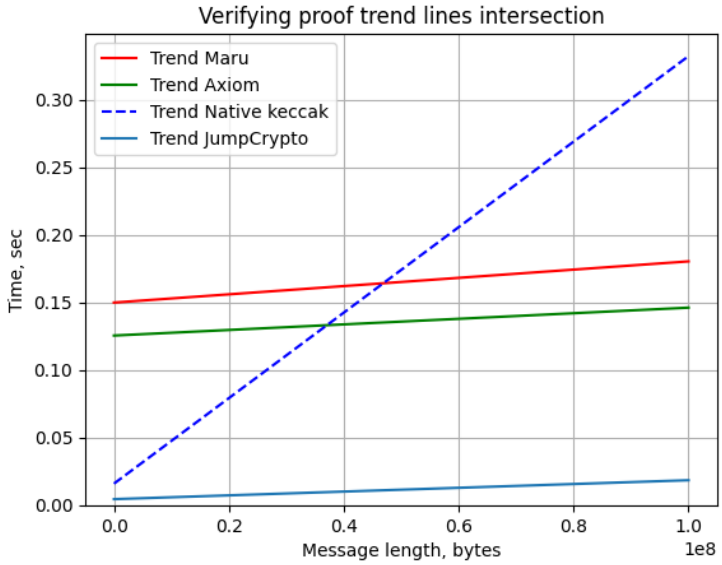

Figure 5: Comparing effectiveness of native keccak with implementations