The benchmarking results for computational integrity with Keccak256 circuits

July 2023

Abstract

This document showcases the benchmarking of various implementations of Keccak-256 computational integrity using different frameworks and zero-knowledge protocols (SNARK, STARK). It was created for the purpose of familiarizing oneself with the performance of existing implementations. Document focuses on Keccak-f[1600] which has 24 rounds. At each input are 5×5 lanes of 64 bit words indicate the lane and the length of each lane is 64-bit word.

Implementations:

Maru using starky and plonky2: link isn't available yet.

Axiom using halo2-lib:
`https://github.com/axiom-crypto/halo2-lib/tree/community-edition/hashes/zkevm-keccak`

JumpCrypto using plonky2:
`https://github.com/JumpCrypto/plonky2-crypto/blob/main/src/hash/keccak256.rs`

Contents

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. The sponge construction is an iterative approach used to create a variable-length input function F with an arbitrary output length. It relies on a fixed-length permutation (or transformation) **f** that operates on a fixed number of bits, denoted as **b**. The width plays a significant role in this construction. The state of the sponge construction consists of **b** = **r**+**c** bits, where **r** represents the bit-rate and **c** represents the capacity. To begin, the input string undergoes reversible padding and is divided into blocks of **r** bits. The **b** bits of the state are initialized to zero. The sponge construction then proceeds in two phases:

- Absorbing phase: during this phase, the **r-bit** input blocks are XORed with the first **r** bits of the state, alternating with applications of the function **f**. This process continues until all input blocks have been processed.
- Squeezing phase: in this phase, the first **r** bits of the state are outputted as blocks, interleaved with applications of the function **f**.

Permutations. The block transformation **f** is a permutation that uses XOR, AND and NOT operations. It is defined for any power-of-two-word size. The basic block permutation function consists of 24 rounds of five steps:

- Compute the parity of each of the 5-bit columns, and exclusive-or that into two nearby columns in a regular pattern.
- Bitwise rotate each of the 25 words by a different triangular number 0, 1, 3, 6, 10, 15, ...
- Permute the 25 words in a fixed pattern.
- Bitwise combine along rows, using $x \leftarrow x \oplus (\neg y \wedge z)$.
- Exclusive-or a round constant into one word of the state.

2 Implementations description

2.1 Implementation by Maru using plonky2 and starky.

The main idea of this implementation to prove the Keccak computational integrity using recursive STARK verification implemented on starky2, which is the part of plonky2 and aggregate sponge construction, permutations proofs to SNARK. Plonky2 is a SNARK implementation based on techniques from PLONK and FRI, including starky - a highly performant STARK implementation. To avoid the difficulties associated with elliptic curve cycles, Plonky2 uses FRI, which support any prime field with smooth subgroups. Regarding to plonky2 implementation we can achieve a smaller recursion threshold of 2^{12} gates using a PLONK arithmetization with custom gates tailored to the verifier's bottlenecks. The circuit operates over a prime field with a modulus of $p = 2^{64} - 2^{32} + 1$. The algorithm comprises multiple parts, involving the generation of the sponge construction and independent proofs for permutations using the STARK scheme. The compatibility between these components is validated through cross-table lookups (CTL). The two validated proofs are aggregated into a single SNARK proof.

2.2 Implementation by Axiom using halo2-lib

Halo2 is built on the SNARK scheme, which includes a PLONKish arithmetization, meaning that it supports custom gates and lookup arguments. Axiom use the Halo2 proving system alongside the KZG polynomial commitment scheme. In all zero-knowledge (ZK) circuits, a one-time universal trusted setup, known as a powers-of-tau. The Axiom circuits are larger and require a larger setup than the one used for EIP-4844. They use the existing perpetual powers-of-tau used in production by Semaphore and Hermez.

The scheme operates over a scalar field BN256 and involve the generation of the sponge construction and independent proofs for permutations using the SNARK scheme with lookups.

The circuit uses Keccak-specific CellManager, which lays out Advice columns in FirstPhase. They form a rectangular region of $Cell\langle F\rangle$'s. Rows records the current length of used cells at each row. New cells are queried and inserted at the row with the shortest length of used cells. The `start_region` method pads all rows to the same length according to the longest row, ensuring that all rows start at the same column.

Absorb. Following padding, in each chunk, further divide this chunk into 17 sections with a length of 64 for each section, pack into keccak-sparse-word absorb. Then compute $A[i][j] \leftarrow A[i][j] \oplus absorb = pack(\&chunk[idx \times 64..(idx+1) \times 64])$ for each of the first 17 words $A[i][j]$, where $i+5j < 17$ and $idx = i+5j$ to obtain output $A[i][j]$. Initialize from all $A[i][j] = 0$ and perform this absorb operation before the 24 rounds of Keccak-f permutation. After the 24-rounds of Keccak-f permutation, then move to the next chunk and repeat this process until the final chunk (which may contain padding data).

Squeeze. At the end of the above absorb process, take $A[0][0]$, $A[1][0]$, $A[2][0]$, $A[3][0]$ to form 4 words, each word is 64-bit in big-endian form, but in the form of Keccak-sparse-word representation. Then unpack them into standard bit representation. The original Keccak hash output would then be the 256-bit word $A[3][0]\|A[2][0]\|A[1][0]\|A[0][0]$ (in big endian form). We divide each of the above 4 words into 8-bit byte representation. This gives 32 bytes $A[0][0][0..7]$, $A[1][0][0..7]$, $A[2][0][0..7]$, $A[3][0][0..7]$, each representing the word in little-endian form. Then RLC using Keccak challenge to obtain $hash_{rlc} = A[0][0][0] + A[0][0][1] \cdot challenge + \ldots + A[3][0][7] \cdot challenge_{31}$, which is the output of the Keccak hash.

Padding. Since NUM_BITS_PER_BYTE = 8, then $RATE\_IN\_BITS = 17 \times 8 \times 8 = 1088$. So pad input bytes (turn into bits first) with $10\ldots01$ in bits until the length in bits reaches an integer multiple of RATE_IN_BITS. The result is divided into chunks of size RATE_IN_BITS.

Keccak table. The Keccak table, used internally inside the Keccak circuit, is a row-wise expansion of the original Keccak table which contains 4 advice columns:

- is_final: when true, it means that this row is the final part of a variable-length input, so corresponding output is the hash result. The external Keccak table has this column always true, but in the internal table used by the Keccak curcuit it is bytewise "expanded", so only true at the last byte of input.
- *input_rlc*: each row takes one byte of the input, and RLC it to add to the previous row *input_rlc*. At the row for the final byte of the input (so *is_enabled* true), this row is the *input_rlc* listed in the external Keccak table.
- *input_len*: similarly, to *input_rlc*, it adds up 8 bits for each row until the final byte which gives the actual input length of the external Keccak table.
- *output_rlc*: it is zero for all rows except the row corresponding to the final byte of input, where result is the RLC of Keccak squeeze. At the final byte row, value is the same as the external Keccak table *output_rlc*.

2.3 Implementation by JumpCrypto using plonky2

The benchmarking objective is to verify the Keccak hash function computational integrity using a SNARK scheme implemented with plonky2 in Rust. Comparing with Maru implementation, implementation by JumpCrypto is a simpler without difficult structures as CTL. The implementation of circuit was built also in Goldilocks field and using limbs as a base tool for proving circuit. The main operations were implemented in $hash\_function\_f1600$ that includes permutations, which are used in proving hash in $hash\_keccak256$ function. This function includes squeezing and absorbing data operating for each block of witness input.

3 Benchmarking methodology

3.1 Criteria

The circuits were tested using various criteria to ensure effectiveness and reliability. The benchmarking process aimed to assess the following criteria for implementations:

Maru implementation.

Check how small the proof size of sponge, permutations, aggregation.
Check the optimal execution speed limits:

- Generation of sponge construction proof for message.
- Generation of permutations proof for message.

- Generation of aggregated proof for sponge construction and permutations proofs.
- Verification of proof for sponge construction.
- Verification of proof for permutations proof.
- Building of circuit for aggregation.
- Verification of aggregated proof for sponge construction and permutations proofs is valid.

Axiom implementation.
Check how small the proof size of Keccak CI verification.
Check the optimal execution speed limits:

- Generation of parameters in BN256 field for KZG commitment.
- Generation of private key and verifier key in trusted setup.
- Building of circuit for message.
- Generation of the Keccak proof for message is valid.
- Verification of the Keccak CI proof is valid.

JumpCrypto implementation
Check how small the proof size of Keccak verification.
Check how many numbers of blocks are needed depending on message length.
Check the optimal execution speed limits:

- Building of circuit for message.
- Generation of the Keccak proof for message is valid.
- Verification that the proof of the Keccak integrity proof for message is valid.

3.2 Indicators

During the benchmarking process, various indicators were employed to assess the performance and effectiveness of the scheme. The following indicators were used:

Message length variation: the circuit was tested with different message lengths to evaluate its ability to handle messages of varying length. This indicator helped determine the scalability and efficiency of the scheme when processing messages of different lengths.

Measurement of the time required to build circuit.

Maru implementation

Proof generation time:

- Measurement of the time required to generate STARK proof of sponge construction.
- Measurement of the time required to generate STARK proof of permutations.
- Measurement of the time required to generate native Keccak generation.
- Measurement of the time required to generate aggregated SNARK proof.

Proof verification time:

- Measurement of the time required to verify sponge construction proof.
- Measurement of the time required to verify permutations proof.
- Measurement of the time required to verify aggregated SNARK proof.

Proof size:

- Measurement of the size of generated proofs for sponge construction proof.

- Measurement of the size of generated proofs for permutations proof.
- Measurement of the size of generated proofs for aggregated proof.

Axiom implementation

Proof generation time:

- Measurement of the time required to generate SNARK proof.

Proof verification time:

- Measurement of the time required to verify SNARK proof.

Parameters and trusted setup generation for circuit:

- Measurement of the time required to generate params in BN256 field for KZG commitment.
- Measurement of the time required to generate private and verifier keys for trusted setup.

Proof size:

- Measurement of the size of generated proofs for Keccak integrity proof.

JumpCrypto implementation

Proof generation time:

- Measurement of the time required to generate SNARK proof.

Proof verification time:

- Measurement of the time required to verify SNARK proof.

Proof size:

- Measurement of the size of generated proofs for Keccak integrity proof.

Measurement of number of blocks needed to fit the hash input into limbs ($target.num\_limbs() >= limbs.len()$).

3.3 Benchmarking hardware

For benchmarking purposes, We used an AWS server with hardware Xeon E5-2697v4, 128GB DDR4 2400MHz on Linux. It is important to note that the projects use a nightly build of Rust and in newer versions, certain modules may be unavailable. The specific version details are as follows: Linux Version: Ubuntu 22.04.2 LTS

4 Benchmarking

Initially, we should bench the native Keccak generation before comparing it with bench using frameworks. The message length being tested ranges from 136 bytes to 100,000,00 bytes. The expected outcome is that the execution time increases linearly based on the input message length. The rate in Keccak-256 is 1088 bits, then it is more favorable to choose a message length that is a multiple of 136 bytes. This is because the rate represents the number of bits that are absorbed into the sponge construction at each step. By aligning the message length with the rate, we can ensure efficient and optimal absorption of the message into the Keccak. For implementations by Maru and Axiom a message length range was used from 136 to 136,000 and from 136 to 19,992 bytes for JumpCrypto implementation due to out of memory. For Axiom implementation a PLONKish circuit depends on a configuration:

- The number of rows n in the matrix. n must correspond to the size of a multiplicative subgroup (a power of two). N equals to 2^k, where k is configuration parameter.
- KECCAK_ROWS is another keccak-specific circuit configuration parameter: the keccak_f sponge permutation has multiple rounds. The circuit is broken up into these rounds, and each round uses KECCAK_ROWS rows of the circuit. It also affects how many columns are allocated for keccak_f.

Depending on the configuration parameters, the capacity of Keccak can be determined, which measures how many rounds are possible in the circuit. For benching, three sets of parameters were chosen that affect the configuration, proof size, proof generation time, and proof verification time:

1. k = 14, keccak_rows = 28
2. k = 15, keccak_rows = 25

4.1 Maru implementation on Linux

The first step is to generate sponge proof and it's expected generation time will increase linearly multiplied by a constant. The average value for this range varies between 0.02 and 0.08 seconds. Using STARK scheme, verification must be fast, close to constant value and remains within the range of 0.03 seconds. The outliers present on the diagram do not significantly affect the verification time. The diagram of proof size has step-like pattern, varies with the length of the message. After proving the correctness of the sponge construction, the next step is to generate proof for the permutations, where proof generation diagram has step-like pattern. There is present correlation between the size of the permutation proof and the time it takes to generate the proof. After The verification time of permutations should remain within a constant time frame of approximately 0.12 seconds. The size of this proof is larger than the size of the proof for the sponge construction, specifically more than four times larger. After generating and verifying two independent proofs within the STARK scheme, they need to be aggregated into the SNARK scheme to generate the final proof. Initially, a circuit with public inputs must be constructed to facilitate the proof generation process. Building the circuit exhibits a relatively constant increasing pattern and does not follow a linear trend. Next, we can generate an aggregated proof that combines the proofs for the aggregation of the sponge and permutations. The generation of the proof is fast and executes in approximately in range from 0.7 to 0.12 seconds, which is excellent compared to the generation of other proofs. The execution time for verifying the aggregated proof is fast, averaging 0.0044 seconds, which aligns with the expected constant value. The proof size of the aggregation remains constant and does not change, indicating that the execution of the generation process will not significantly increase constant.Also, in this analysis, it is necessary to consider the circuit construction time of sponge&permutations in STARK, aggregation to SNARK. The circuit construction time of sponge STARK exceeds in range from 0.0001 to 0.014, which depends of message length as input, which in trace generation padded to pow of 2. The same situation we can expect in circuit construction of permutations, where time execution exceeds in range from 0.01 to 5.3 seconds. Finally, the circuit construction time for aggregating two STARKs is approximately a constant average of 0.004 seconds. **To summarize.** After analyzing each part of the STARK scheme for the Keccak function on Linux platforms, for message lengths that are multiples of 136 within the range of 136 to 136,000 bytes, it can be generally concluded that the overall execution time of the scheme has great perfomance. The generation time of the permutation proof played a crucial role in this conclusion. Additionally, it should be noted that the native Keccak generation is approximately 350 times more efficient than the verification of the permutation proof for the specified benchmarking range.

4.2 Axiom implementation with parameters k = 14, keccak_rows = 28 on Linux

The Keccak capacity for these parameters is equal to 21, which means there are 21 possible rounds in the circuit. The generation time for KZG (Kate-Zaverucha-Goldberg) commitment parameters remains constant and is included in the circuit construction time along with the generation of the trusted setup. The circuit construction is performed quickly and, on average, takes 0.85 seconds, considering the determination of the number of rows, which is equal to 2^k. The generation of the trusted setup is fast and remains constant, with an average generation time of 0.42 seconds. The generation time of the proof increases slowly but remains close to constant. On average, proof generation takes 5.6 seconds. For this range of message lengths, the size of the proof remains constant at 50,176 bytes, which is relatively low for a SNARK scheme. Proof verification runs quickly and does not significantly increase depending on the message length. On average, proof verification takes 0.188 seconds.

4.3 Axiom implementation with parameters k = 15, keccak_rows = 25 on Linux

This variant is the most time-consuming because the number of possible rows has significantly increased compared to the previous parameters, amounting to 50. The parameter 'k' is equal to 15, which causes the time required for generating the KZG parameters and building the circuit to increase by half, taking 1.38 seconds. The generation of the trusted setup has an average time of 0.58 seconds. At the beginning, we mentioned that this is the most resource-intensive parameter variant for benchmarking among those selected. This is due to the fact that the proof

generation time is more than twice as long compared to the previous parameter variant in terms of time and 10% times larger in terms of proof size. On average, proof generation takes 10.6 seconds and remains relatively constant. The size of the proof remains the same as in the previous parameter variants for this range of message lengths, at 52,704 bytes. Verification occurs more than twice as quickly compared to previous parameters, with an average time of 0.125 seconds. If the verification time of the proof remains constant despite an increase in message length, it suggests that the verification process is not affected by the message length, thus keeping the verification time constant.

To summarize. there is no specific approach to parameter selection. The selection of parameters depends on your specific task. In general, when k is smaller, the generation of the proof is faster, but the verification process becomes more computationally expensive. When selecting a fixed value for k, it is recommended to determine the precise number of keccak_f functions your circuit will utilize and set the keccak_rows parameter to be as large as possible while still fitting within 2^k. rows. The generation of KZG parameters and the generation of the trusted setup depend on the parameter k. For the selected range in benchmarking, the proof generation time increases slowly, but it will continue to increase for very large lengths. The proof size depends on the capacity and remains constant for any message length, while the proof verification time is also constant.

4.4 JumpCrypto implementation on Linux

The message length range was chosen according to device's specifications. The benchmarking was produced for message length ranging from 136 to 19,992 bytes. To perform the benchmarking, you need to input the message length, the message hash for verification, and the number of blocks. Depending on the message length, the number of blocks needs to be increased. The number of blocks increases linearly with the message length. During benchmarking, a condition was chosen that if the iteration number is greater than twice the number of blocks, it should be increased by 4. The circuit building time for proof takes much longer compared to other implementations and depends on the number of blocks corresponding to the message length, which increase linearly. Within the message length from 1000, where the number of blocks increases, the change does not increase linearly. If we tested for a bigger range, we would notice a significant increase in the time required for circuit building and proof generation corresponding to the increase in the number of blocks. The diagrams of proof generation and circuit building have almost identical patterns, but perfomance of circuit building is lower than generation one. However, there is a time difference, and it favors the proof generation. Depending on the number of blocks, this time will also increase linearly. Compared to the Maru implementation, which is also implemented on Plonky2, the proof size changes according to a step-like pattern plot. However, the initial proof size of this implementation is twice as large as the previous one. Verification takes place in constant time, averaging 0.0051 seconds, and there are no significant time increases for a small range.

To summarize. proof generation and circuit building are nearly equivalent in time. There is a strong dependency on the number of blocks and the input message for the correctness of this implementation. The number of blocks for this range, ranging from 4 to 152, with message sizes from 136 to 19,992 bytes, will increase as the length of the message increases. The time required for circuit building and proof generation increases almost linearly but depends on the message size and the number of blocks. The proof size ranges from 137,268 to 171,592 bytes, which is more than twice as large as in previous implementations. Verification remains constant within this range.

5 Comparing results

We will compare implementations by Maru, Axiom, JumpCrypto. The selected parameters for comparison are circuit building time, proof generation time, proof size, and 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, k = 15, and keccak_rows = 25. With these parameters, we will have 50 possible rounds in the circuit.
- JumpCrypto implementation with input message length ranging from 136 to 19,992 bytes.

Building circuit. 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, where the time is lower to 48,000 bytes than Axiom implementation and better overall than JumpCrypto. Next in terms of speed is the Axiom implementation, where the average building time is approximately 1.38 seconds.

The JumpCrypto implementation shows slower performance due to its dependence on the number of blocks for the input message length and curve displayed on the plot was transformed into a logarithmic curve, while the initial dataset exceeds 10744 bytes in just 220 seconds. As a result, the circuit building time is very high.

Proof generation. Axiom implementation shows the best results of proof generation for this range of message length. However, considering that blockchain hashes, particularly from Ethereum, have sizes up to 10KB, the Maru implementation demonstrates better performance with sizes up to 90KB. The JumpCrypto implementation show faster perfomance than Axiom with message input size up to 2448 bytes, then the time increases with number of blocks faster than other implementations. If you prioritize larger message lengths in bytes, then the Axiom implementation can be suitable for your needs. However, 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. Overall, 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 implementation is suitable 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 verification. 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.

6 Summary

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.

7 Appendix

7.1 Native Keccak generation

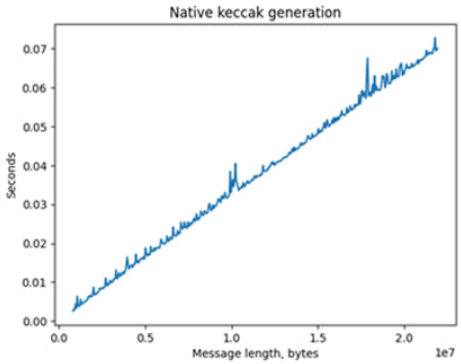
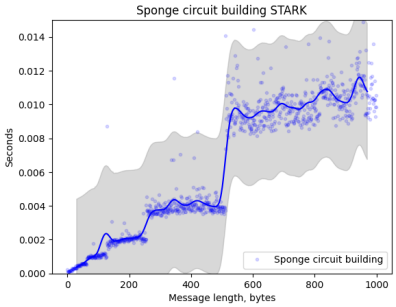
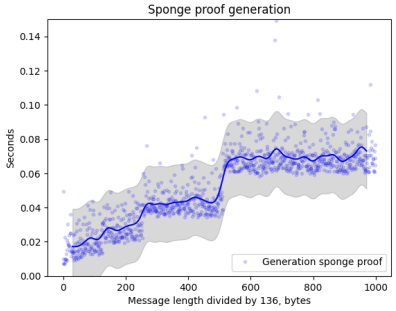


Figure 1: Image 2: Native Keccak generation in range up to 21 MB

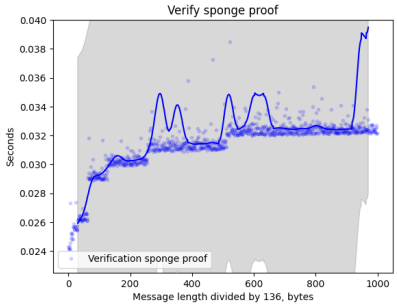
7.2 Maru Implementation



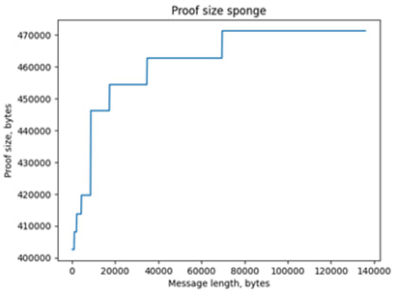
(a) Image 1: Sponge circuit building



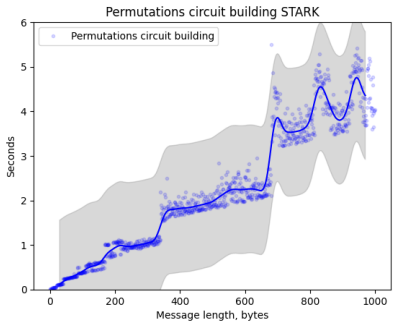
(b) Image 2: Generation of sponge proof



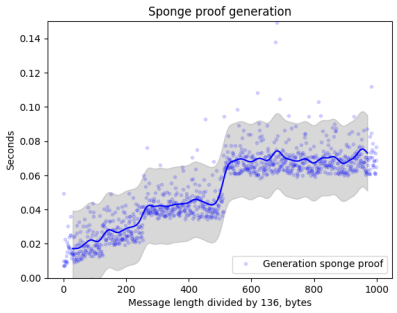
(c) Image 3: Verification of sponge proof



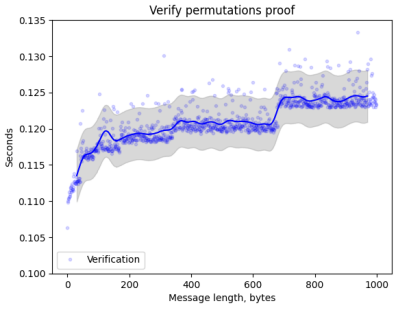
(d) Image 4: Proof size of sponge



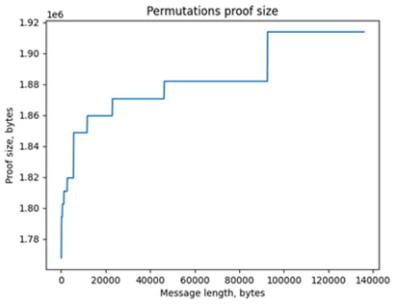
(e) Image 5: Permutations circuit building



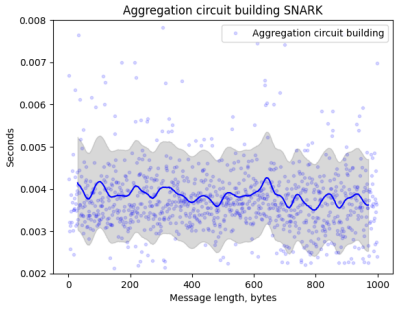
(f) Image 6: Permutations proof generation



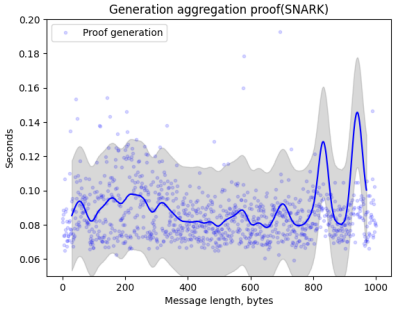
(g) Image 7: Permutations proof verification



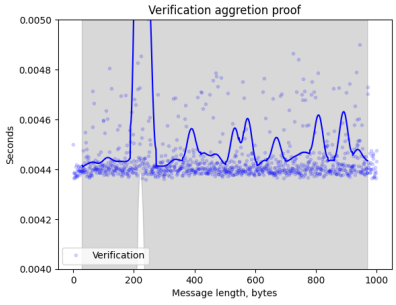
(h) Image 8: Proof size of permutations



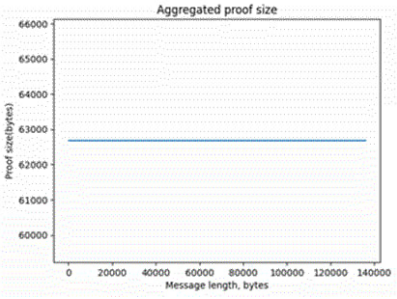
(i) Image 9: Building aggregation circuit



(j) Image 10: Generation of aggregation proof



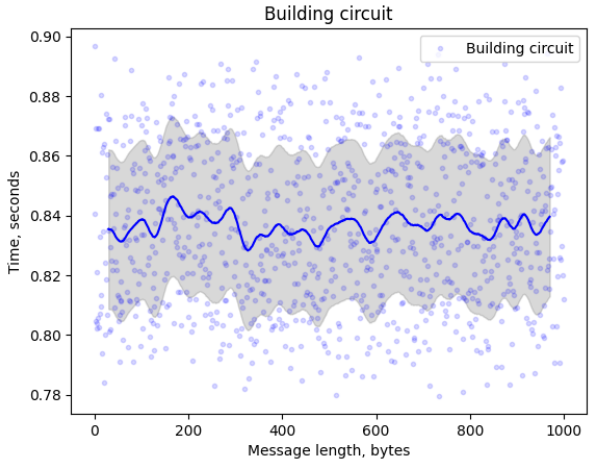
(k) Image 11: Verification of aggregation proof



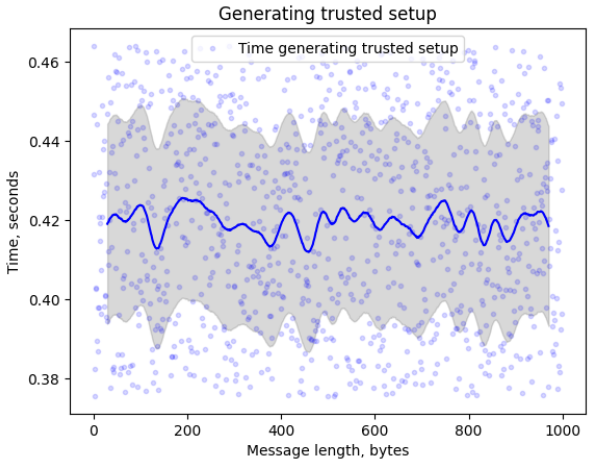
(l) Image 12: Aggregation proof size

Figure 2: Maru Bench

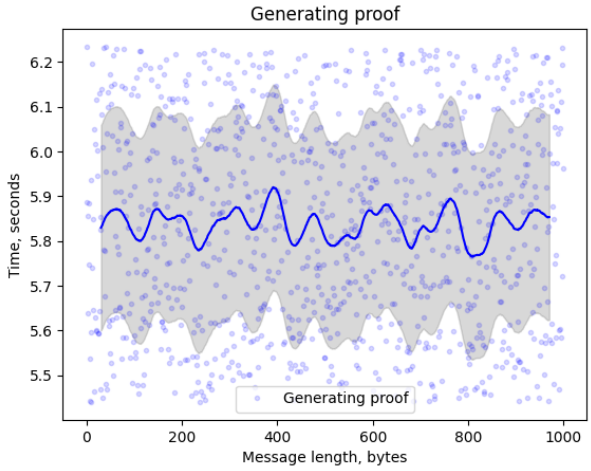
7.3 Axiom implementation



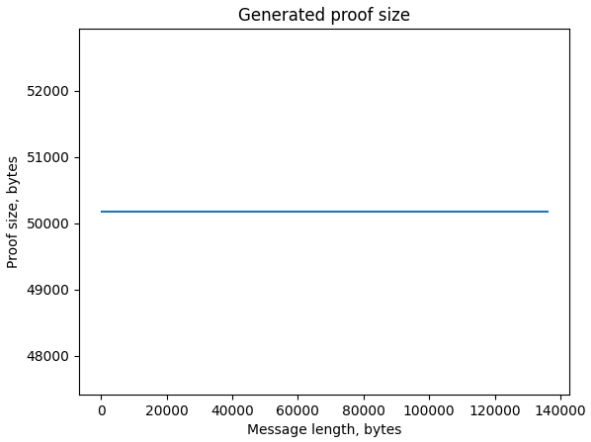
(a) Image 1: : Circuit building



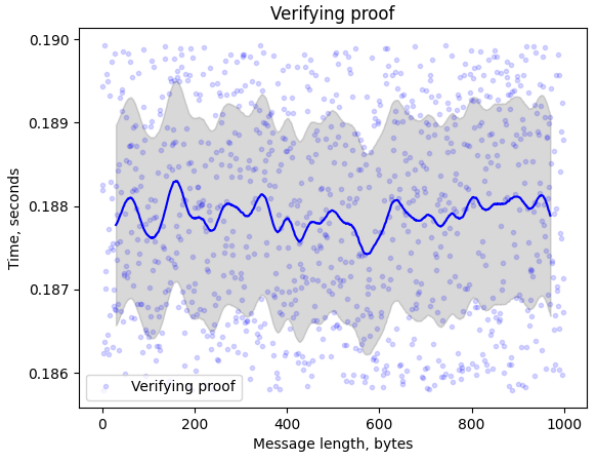
(b) Image 2: Trusted setup generation for circuit



(c) Image 3: Keccak proof generation



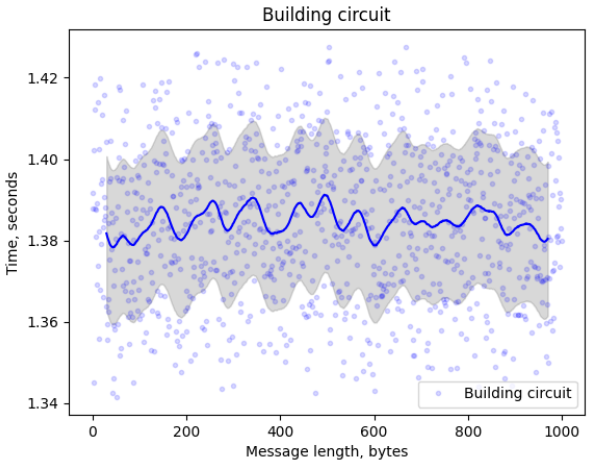
(d) Image 4: Proof size



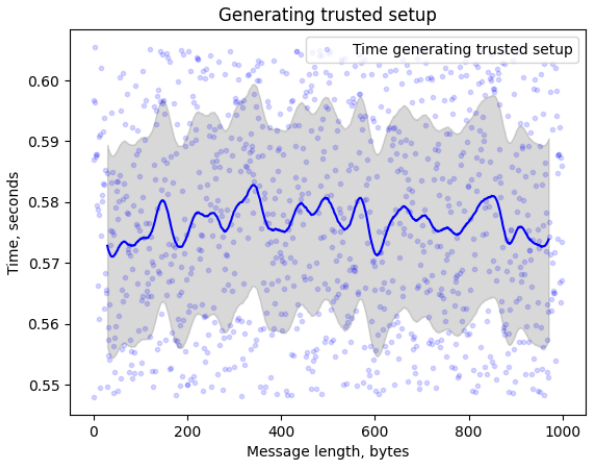
(e) Image 5: Proof verification

Figure 3: Axiom bench with params k=14, keccak_rows=28

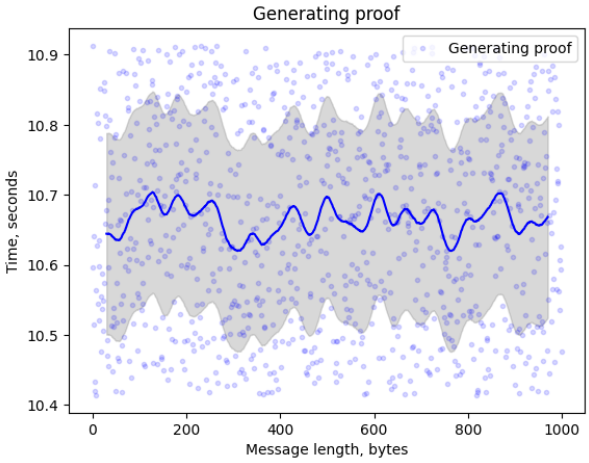
7.4 Axiom implementation



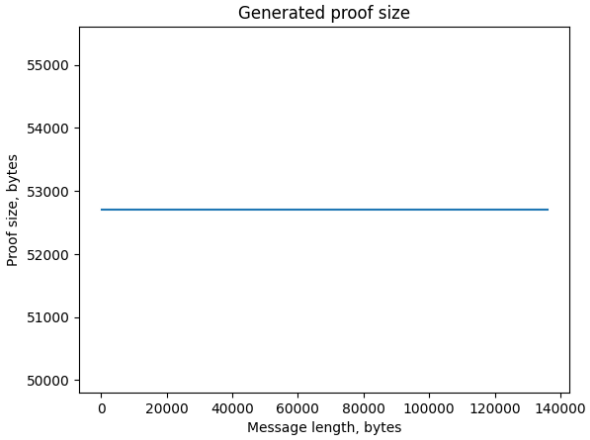
(a) Image 1: : Circuit building



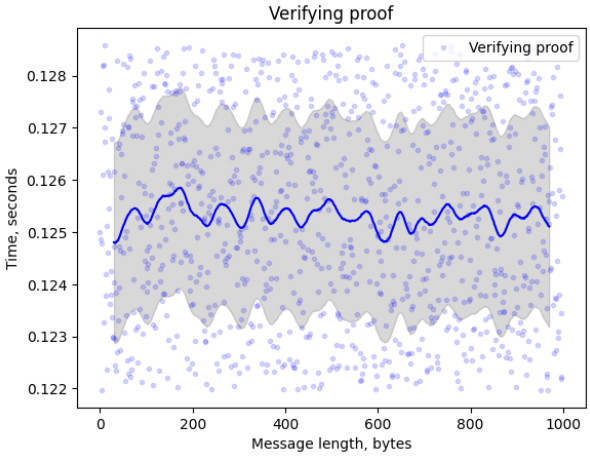
(b) Image 2: Trusted setup generation for circuit



(c) Image 3: Keccak proof generation



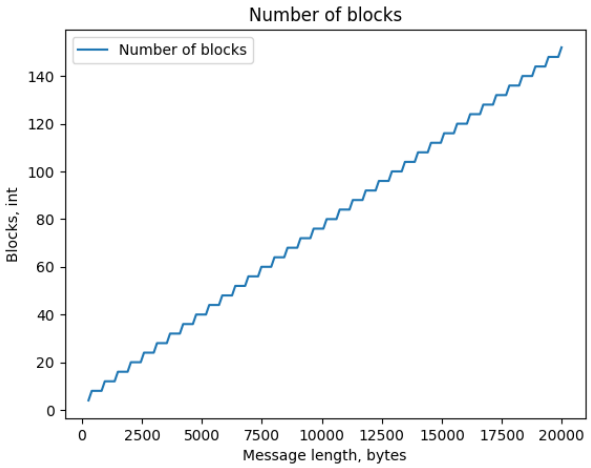
(d) Image 4: Proof size



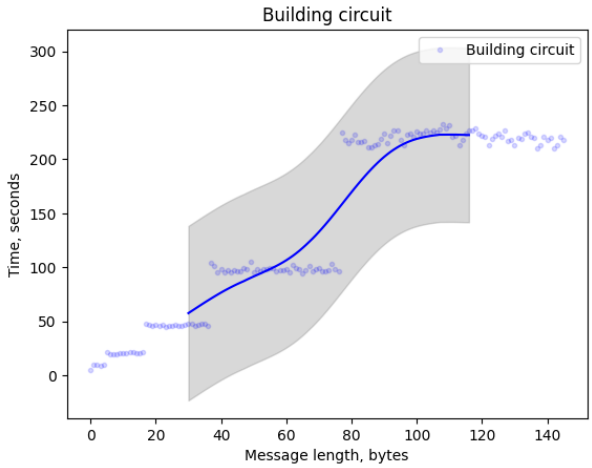
(e) Image 5: Proof verification

Figure 4: Axiom bench with params k=15, keccak_rows=25

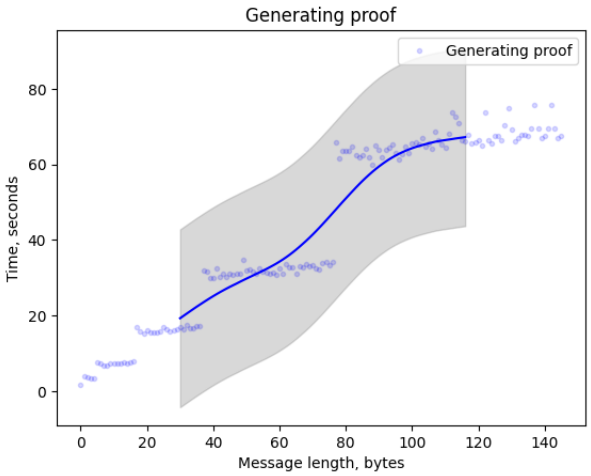
7.5 JumpCrypto implementation



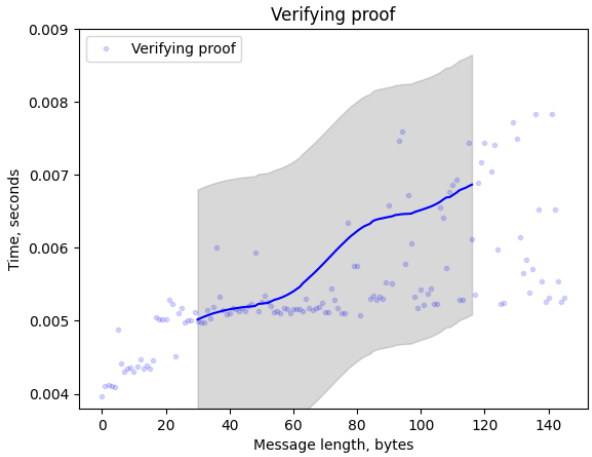
(a) Image 1: Increasing number of blocks



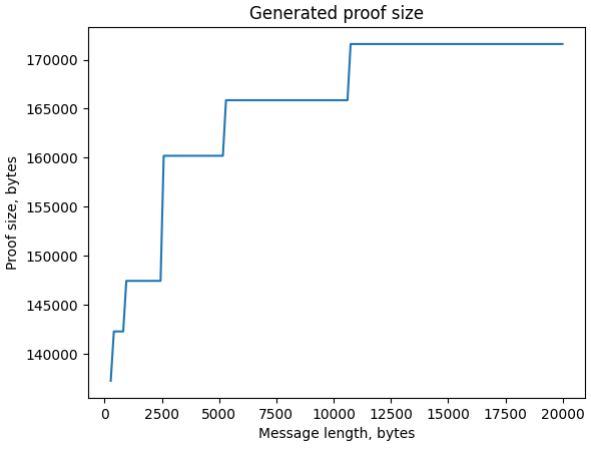
(b) Image 2: Circuit building



(c) Image 3: Keccak proof generation



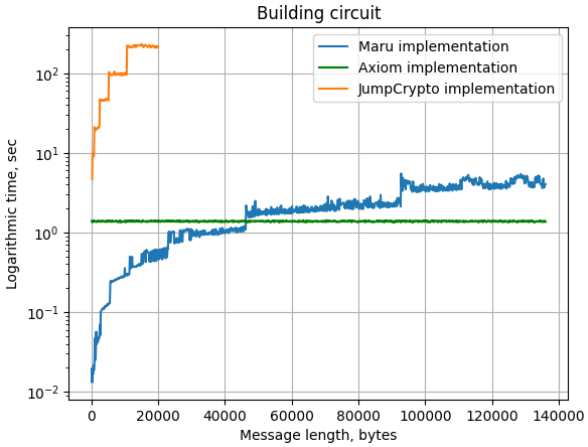
(d) Image 4: Proof verification



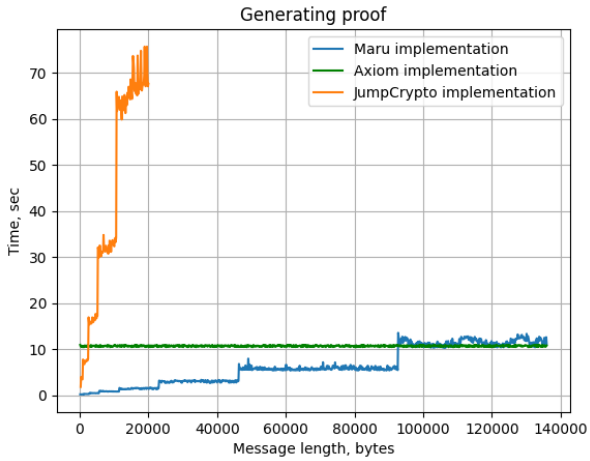
(e) Image 5: Proof size

Figure 5: JumpCrypto bench

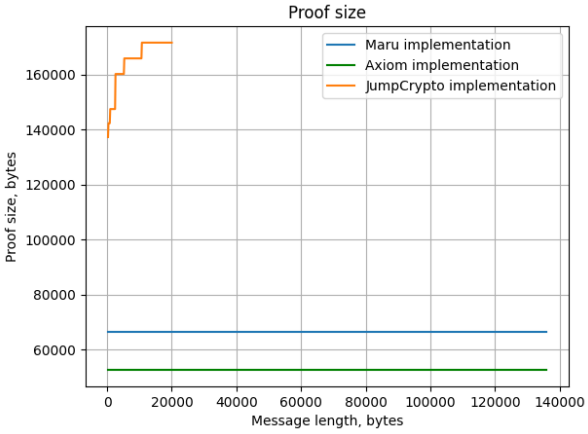
7.6 Comparison of Maru and Axiom implementations



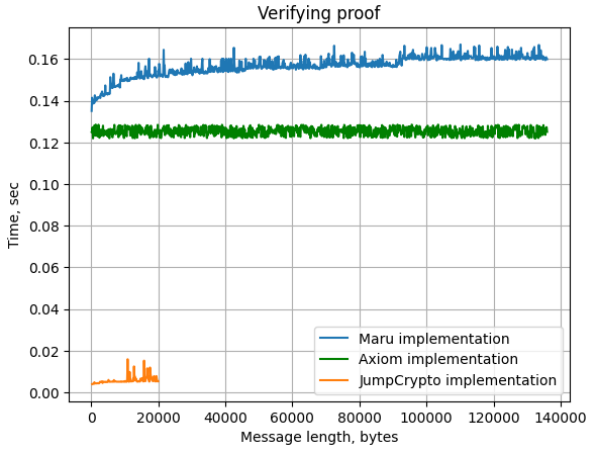
(a) Image 1: building circuit time using logarithmic scale function



(b) Image 2: Proof generation



(c) Image 3: Proof size



(d) Image 4: Proof verification

Figure 6: Comparison of Maru and Axiom

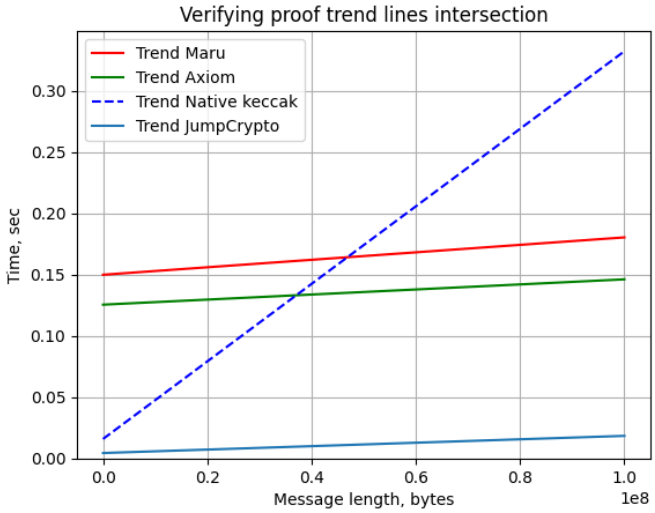


Figure 7: Effectiveness of implementations comparing with native keccak verification