The benchmarking results for computational integrity with Keccak256 circuits

July 2023

Abstract

This document showcases the testing of various implementations of computational integrity of Keccak-256 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:

1. Maru using starky and plonky2 – link wasn't published yet.
2. Axiom using halo2-lib - `https://github.com/axiom-crypto/halo2-lib/tree/community-edition/hashes/zkevm-keccak`
3. 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, b, plays a significant role in this construction. The state of the sponge construction consists of b = r+c bits, where r represents the bitrate 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 computational integrity of Keccak using recursive STARK verification implemented on starky2, which is the part of plonky2 and aggregate sponge construction and permutations proofs to SNARK. Plonky2 is a SNARK implementation based on techniques from PLONK and FRI. It has since expanded to include tools such as starky, a highly performant STARK implementation. Unlike Halo 2, where recursion is implemented without pairings, enabling much smaller elliptic curves. 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, with the initial two 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. The correspondence between the message and its hash is achieved using the STARK scheme, and the resulting proof is encapsulated within the SNARK scheme.

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 utilizes 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 ceremony, is employed. This ceremony is similar to the one being conducted by the Ethereum Foundation in anticipation of EIP-4844. 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 ceremony used in production by Semaphore and Hermez, specifically this challenge.

The scheme operates over a scalar field BN256. The scheme comprises multiple parts, with the initial two parts involving the generation of the sponge construction and independent proofs for permutations using the SNARK scheme with lookups. Unlike the previous implementation, CTL is not used here.

The circuit uses Keccak-specific CellManager, which lays out Advice columns in FirstPhase. They form a rectangular region of Cell$\langle F\rangle$'s. CellManager.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. This is used when a new region is started for witness synthesis.

State data before and after the Keccak-f permutation are stored in s[i][j] and $s\_next[i][j]$ for $0 \leq i, j \leq 4$. Each $s[i][j]$ stands for a 64-bit Keccak-sparse-word-representation (with bits standing for [0, ..., 7]). The final Keccak hash makes use of the sponge function, with arbitrary bit-length input and 256-bit output.

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. $NUM\_WORDS\_TO\_ABSORB = 17$, $NUM\_BYTES\_PER\_WORD = 8$, then $RATE = 17 \times 8$. 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...01 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*.

The multi-packed implementation for Keccak proof uses special arithmetic called sparse-word-representation, this splits a 64-bit word into parts. Each bit in the sparse-word-representation holds BIT_COUNT number of 0/1 bits, so equivalent to BIT_SIZE as the base of bit in the sparse-word-representation. Something need to be said about RLC that used in Keccak table. RLC uses keccak input challenge. Update *data_rlcs* on each of the 17 rounds where we absorb data based on input bytes. This means after each round *data_rlcs*[0] contains the final RLC of input bytes up to this round. This is set to be the *data_rlc* of the current round, and it is passed to the next round

so that after all 17 absorb rounds, we get the final *data_rlc* for the current chunk data. This is then passed to the next chunk until the last byte of input data, where *data_rlc* must exclude padding data. So, in the final part of the input byte *data_rlc* will be equal to the corresponding *output_rlc* in the Keccak table.

2.3 Implementation by JumpCrypto using plonky2

The testing objective is to verify the computational integrity of the Keccak hash function 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 testing 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 testing process, various test indicators were employed to assess the performance and effectiveness of the scheme. The following test 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 Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (4 cores and 8 threads) with 8 GB RAM. The benchmarking was performed using a linear approach on both Windows and Linux platforms. It is important to note that the projects utilized 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

Windows Version: Windows 10 Home edition

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. In comparison, this can help make conclusion about the faster and more efficient verification of permutations or Keccak generation. For implementations by Maru and Axiom a message length range was used from 136 to 136,000 and from 136 to 2488 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 = 25
2. k = 15, keccak_rows = 28
3. k = 15, keccak_rows = 9

4.1 Maru implementation on Windows

The first part of generating proof is generate sponge proof and it's expected that execution time of generation will increasing linearly multiplied by constant. Using diagram, it is evident that the execution time does not increase strictly linearly. However, by using a quadratic trend line, it is possible to fit all the data points for the generation time. The average value for this range varies between 0.1 and 0.3 seconds. The next step after generation is verification of this proof. Using STARK scheme verification must be fast and close to constant. The verification time occurs almost at a constant rate and should remain within the range of 0.3 seconds. The outliers present on the diagram do not significantly affect the execution time. It is also necessary to demonstrate the proof size in bytes for the sponge construction. The diagram of proof size is step-like pattern and varies with the length of the message. The proof size tends to increase at a slower rate. After proving the correctness of the sponge construction, the next step is to generate proofs for the permutations. We have also a step-like diagram, but depending on the message length, the execution time for proof generation can significantly increase. There is present correlation between the size of the permutation proof and the time it takes to generate that proof. After The verification time of permutations should remain within a constant time frame of approximately 0.14 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 0.10 seconds, which is excellent compared to the generation of other proofs. The execution time for verifying the aggregated proof is fast, averaging 0.0050 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.

4.2 Maru implementation on Linux

The generation time of sponge construction has pattern, which similar on Linux, but there are no significant variations in values as observed on Linux. Therefore, the graph will exhibit linear increasing, and the average values for this range will be 0.16 seconds. Verification of proof expected to be constant. Looking at the diagram on Linux, there were outliers up to 0.10 seconds and average range was from 0.03 to 0.04, while here the situation is improved, and the values lower within the range of 0.020 to 0.025 seconds, which is a constant time. The proof size for each computation remains constant, and there is no need to provide a detailed description. The generation of permutation proofs exhibits a similar pattern on both Linux and Windows. However, the speed of generation is higher on Linux, and there are noticeable outliers in the message range from 100,000 to 140,000. It can be concluded that the generation of proofs performs better and faster on Linux compared to Windows. While the message length increases, the execution speed of this component will significantly increase compared to Linux. The verification was expected to be performed within a constant time range of 0.07 to 0.08 seconds, that we can observe on the diagram. It can be concluded that the verification of permutation proofs on Windows is significantly faster than on Linux, and this is a worthy improvement. The point is that the building SNARK circuit is executed approximately three times faster than on Linux, with an average time of 0.004 seconds and almost constant, which is good for our needs. The execution time for generation is very similar to the diagram on Linux but more efficient, approximately 5-10%. While this difference may not be significant compared to other parts of the testing, it still indicates a significant improvement in efficiency. The verification process is constant and takes less time than on Linux, which is a very positive outcome. Additionally, there is an outlier at 0.03, which is not significant in the overall context.

4.3 Maru implementation for extended range on Windows

This part includes previous version of implementation on Windows without aggregation with a larger step to see how the time increases with the length of the message. For this testing, the range started at 1360 bytes, and the initial size was increased by i iterations, where the iterations occurred up to 350,000 bytes, and then increased by a step of 50,136 bytes up to 700,000 bytes. This range was the maximum on my device because there was not enough memory to continue testing, due to computationally expensive cost of generating permutations. The sponge construction proof generation increases depending on the message length within this range, reaching 4 seconds for a message length of 700,000 bytes. This value will increase linearly. The verification of this proof will also increase, but on average, within this range, it will have a value of 0.2 seconds. Regarding the increase in proof size, the diagram should exhibit a step-like pattern like previous tests. As mentioned earlier, the generation time of permutation proofs will significantly increase with larger message lengths on Windows. We can observe from the diagram that for the range starting from 400,000 bytes, the generation time will be around 800 seconds, and this time will increase significantly. The verification of this proof has also increased compared to the range up to 136,000 bytes, by at least a factor of 5. The proof size has increased by only 200,000 bytes, which is excellent for this implementation. As I mentioned earlier, there is a correlation between the proof generation time and the proof size.

To summarize. After analyzing each part of the STARK scheme for the Keccak function on both Windows and 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 was better on Linux. The generation time of the permutation proof played a crucial role in this conclusion. The difference of 10+ seconds is significant compared to other operations that perform better in milliseconds on Windows. 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 testing range. With the increase in the size of the message to 700,000 bytes, the generation time of the permutation proof has significantly increased, reaching around 700-800 seconds. This indicates that it is the most computationally expensive operation in this scheme. Additionally, there has been a slight increase in the execution time of other operations in the scheme, specifically the generation of the sponge proof, which has increased to 5 seconds.

4.4 Axiom implementation with parameters k = 14, keccak_rows = 25 on Linux

The Keccak capacity for these parameters is equal to 24, which means we have 24 possible rounds in the circuit. The generation of parameters for KZG (Kate-Zaverucha-Goldberg) commitment KZG parameters increases with the message length, but there is no clear linear increasing and on average, generation takes 1.65 seconds. The next step is to build the circuit for proving the Keccak hash function. The construction of the circuit occurs quickly, considering the determination of the number of rows, which is equal to 2^k. On average, the build circuit takes 0.000000145 seconds and increases slowly. The generation of trusted setup is fast, but it increases slowly. On average, the generation takes 0.48 seconds. The generation time of the proof increases, but there is no clear increasing pattern. On average, the generation of the proof takes 11.5 seconds. For this range of message length, the size of the proof remains constant at 52,704 bytes, which is relatively low for a SNARK scheme. Proof verification runs quickly and does not significantly increase depending on the message. On average, proof verification takes 0.8 seconds.

4.5 Axiom implementation with parameters k = 15, keccak_rows = 28 on Linux

The Keccak capacity for these parameters is equal to 44, which means we have 44 possible rounds in the circuit. Unlike the first set of parameters, the generation time of KZG parameters has increased. This increase is due to the increasing of the configuration k parameter. On average, the parameter generation process takes 3.35 seconds, indicating a relatively fast generation, but increasing this parameter will result in longer generation times. For this implementation, the maximum value of k can be 17. The building time of the circuit has increased and will vary depending on the value of k. In comparison to the first set of parameters, with an increasing of k, we can observe the generation time of the trusted setup has doubled. However, it still takes an average of 0.875 seconds, indicating a relatively fast generation process. The generation time remains within the range of 0.86 to 0.9 seconds. We can observe that the trend line of degree 2 forms a downward-facing parabola. Although, generation time does not decrease with the message length range of 120,000 bytes. Like the previous test, the size of the proof remains unchanged for this range and is equal to 50,176 bytes. Though, there is no noticeable increasing in verification time. For this range of message lengths, the generation time fluctuates within a small range and averages at 0.123 seconds.

4.6 Axiom implementation with parameters k = 15, keccak_rows = 9 on Linux

This variant is the most time-consuming because the number of possible rows in the loop has increased significantly compared to other parameters and amounts to 144. The parameter k is equal to 15, so the time for generating the KZG and parameters and building circuit remain unchanged. The generation of the trusted setup remains unchanged because k is the same as in the previous variant. On average, the generation takes 0.87 seconds and increases very slowly. At the beginning, I mentioned that this is the most resource-intensive variant of parameters for testing among those that were selected. This is due to the fact that proof generation time and proof size are more than twice as large compared to other variants in terms of time and half times larger in terms of proof size. On average, proof generation takes 41.5 seconds and increases slowly. The size of the proof remains unchanged, as in the previous parameter variants for this range of message length, and it is equal to 67520 bytes. Verification occurs more than twice as fast compared to other parameters, taking an average of 0.054 seconds. If the verification time of the proof remains constant despite an increase in the message length, it suggests that the verification process is not affected by the message length. Therefore, the verification time remains 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 testing, 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.

4.7 JumpCrypto implementation on Linux

The message length range was chosen according to computer's specifications. The testing was produced for message length ranging from 136 to 2448 due to out of memory for selecting a larger range. To perform the testing, 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 testing, 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. 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 0.0035 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 20, with message sizes from 136 to 2448 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 147,452 bytes, which is more than twice as large as in previous implementations. Verification remains constant within this range.

5 Comparing results

We will compare two implementations by Maru and Axiom excluding JumpCrypto implementation because of testing small range of message length. 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 = 14, and keccak_rows = 25. With these parameters, we will have 24 possible rounds in the circuit.

Building circuit. We can see that 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 construction 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.

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 50KB. 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 50KB 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, 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 size range.

Proof verification. In terms of verification efficiency Axiom implementation shows better result, 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 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.

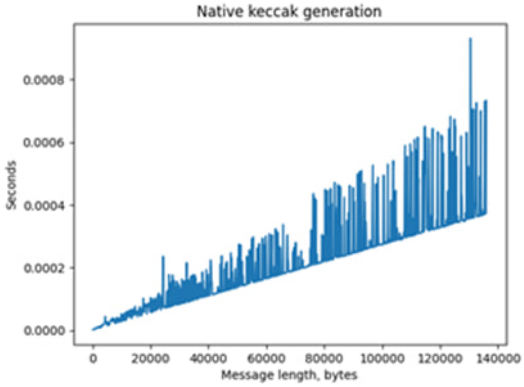
6 Summary

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.

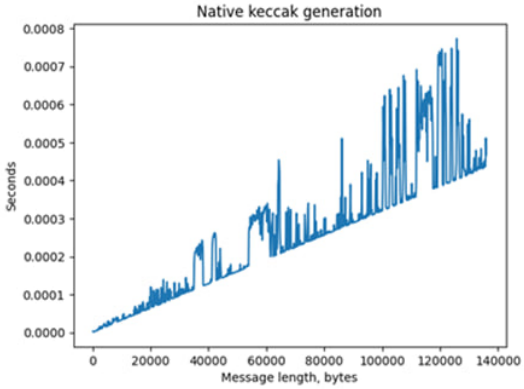
7 Appendix

7.6 Comparison of Maru and Axiom implementations

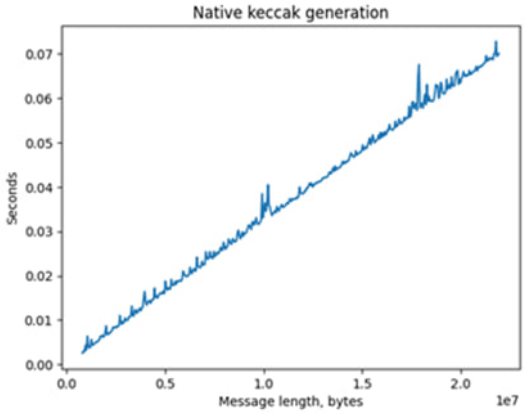
7.1 Native Keccak generation on Windows and Linux



(a) Image 1: Native Keccak generation in range up to 132 KB on Windows



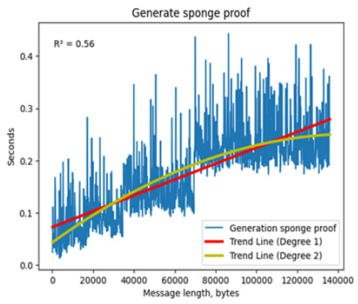
(b) Image 2: Native Keccak generation in range up to 132 KB on Linux



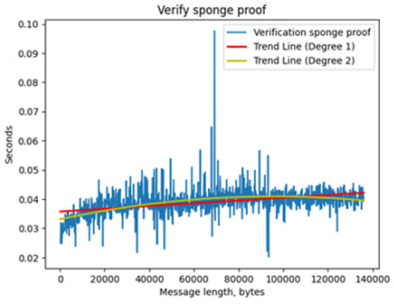
(c) Image 3: Native Keccak generation in range up to 21 MB

Figure 1: Native Keccak generation

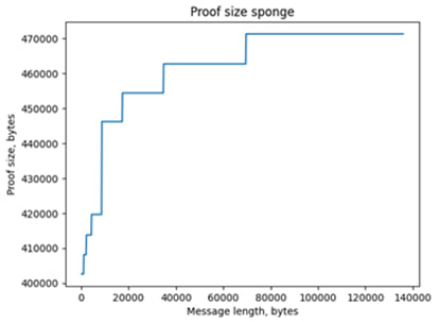
7.2 Maru implementation



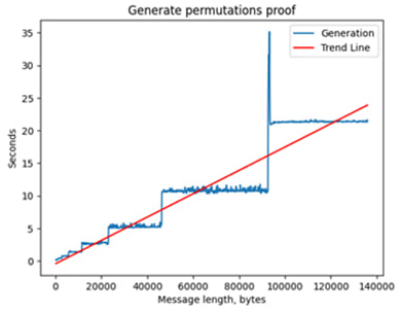
(a) Image 1: Generation of sponge proof



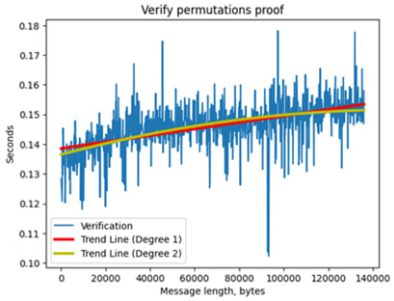
(b) Image 2: Verification of sponge proof



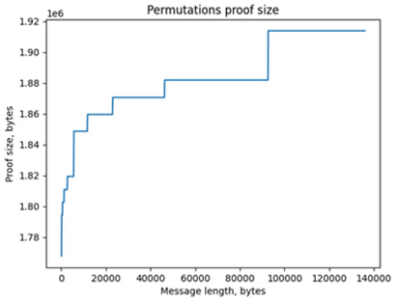
(c) Image 3: Proof size of sponge



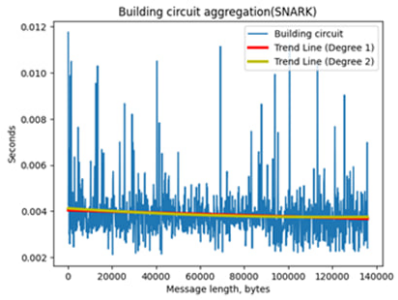
(d) Image 4: Permutations proof generation



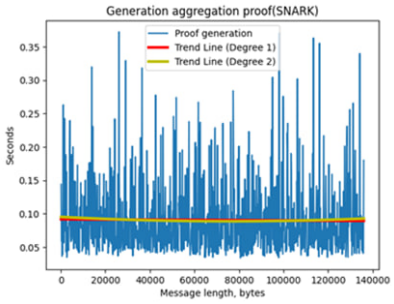
(e) Image 5: Permutations proof verification



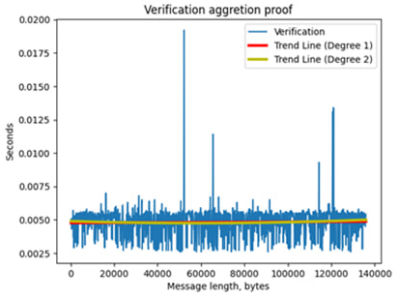
(f) Image 6: Proof size of permutations



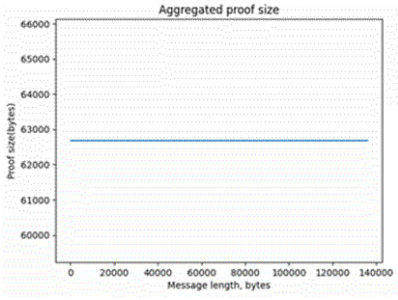
(g) Image 7: Building aggregation circuit



(h) Image 8: Generation of aggregation proof

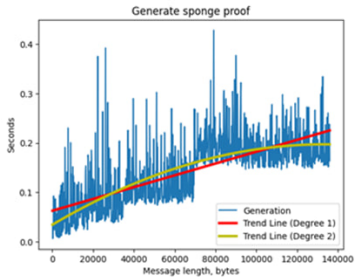


(i) Image 9: Verification of aggregation proof

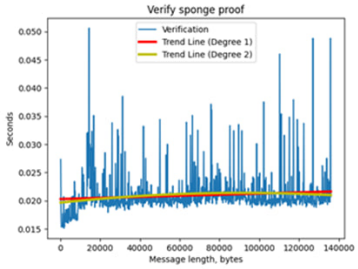


(j) Image 10: Aggregation proof size

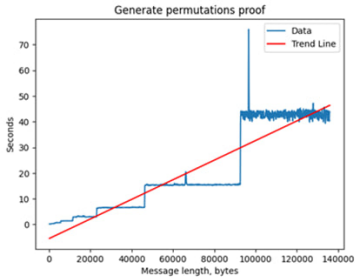
Figure 2: Maru bench on Linux



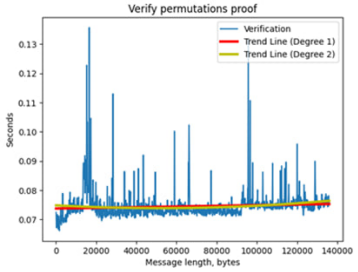
(a) Image 1: Generation of sponge proof



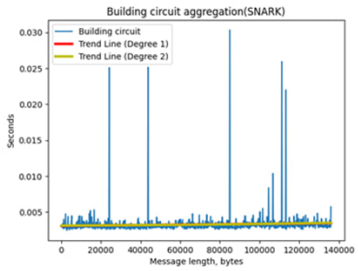
(b) Image 2: Verification of sponge proof



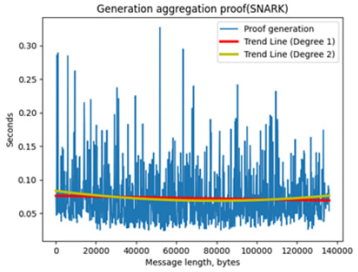
(c) Image 3: Permutations proof generation



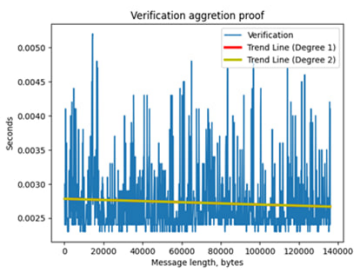
(d) Image 4: Permutations proof verification



(e) Image 5: Building aggregation circuit

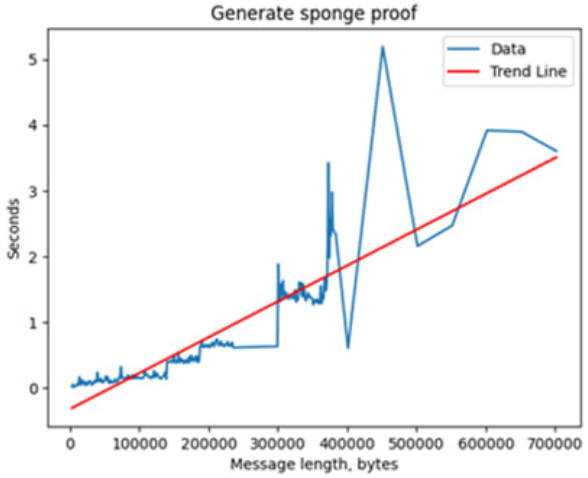


(f) Image 6: Generation of aggregation proof

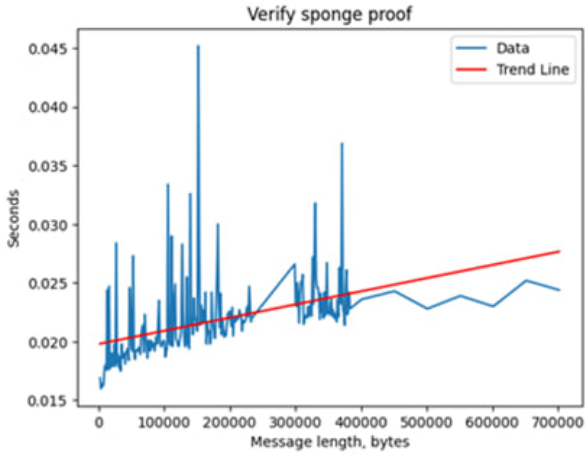


(g) Image 7: Verification of aggregation proof

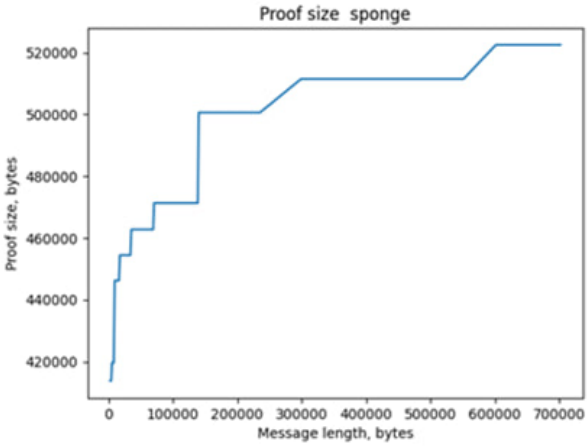
Figure 3: Maru bench on Windows



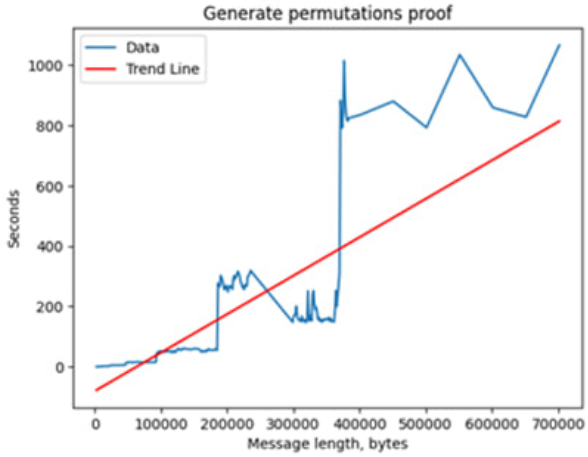
(a) Image 1: Generation of sponge proof



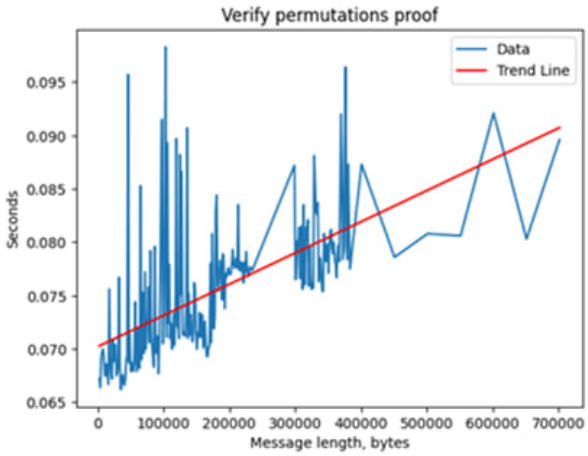
(b) Image 2: Verification of sponge proof



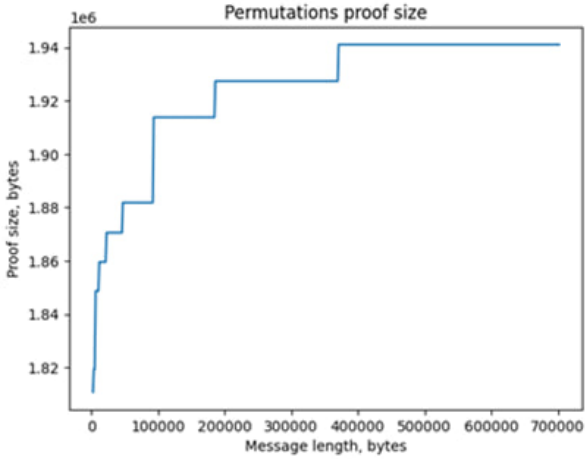
(c) Image 3: Sponge proof size



(d) Image 4: Permutations proof generation



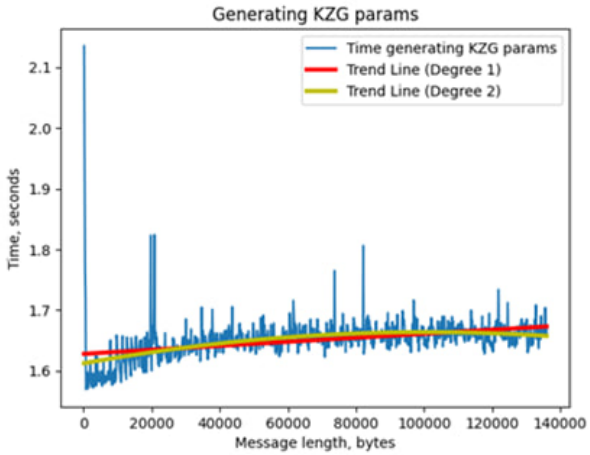
(e) Image 5: Permutations proof verification



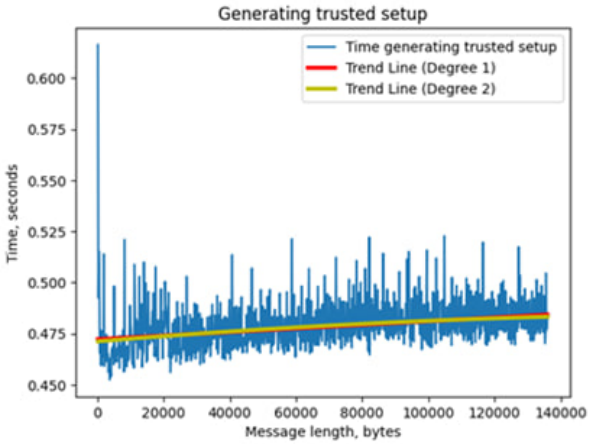
(f) Image 6: Permutations proof size

Figure 4: Maru extended bench on Windows

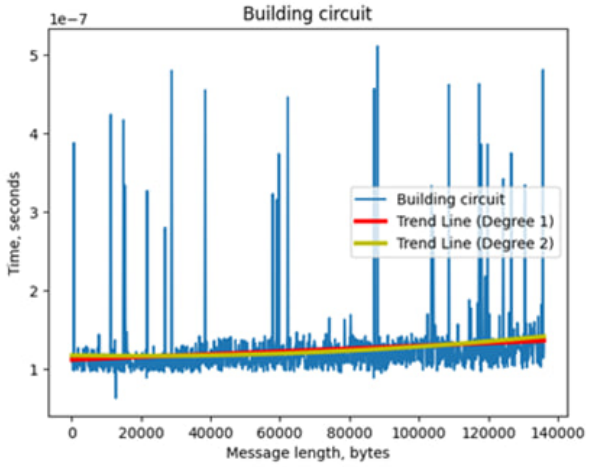
7.3 Axiom implementation



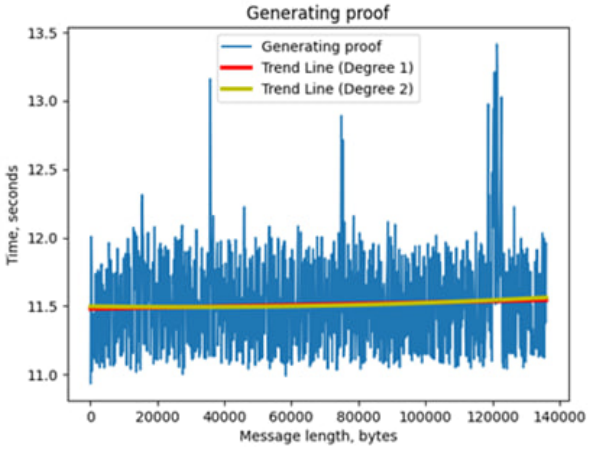
(a) Image 1: : KZG parameters generation



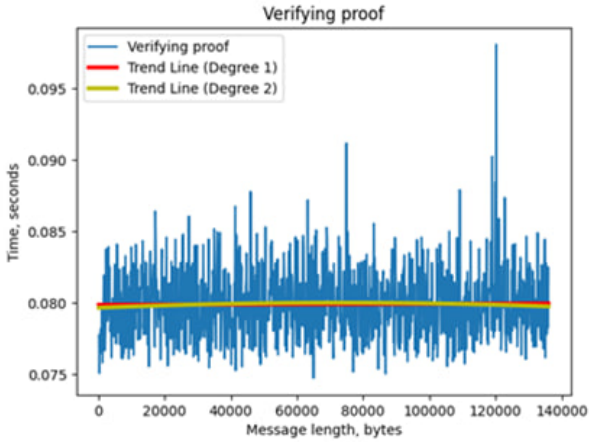
(b) Image 2: Trusted setup generation for circuit



(c) Image 3: Circuit building

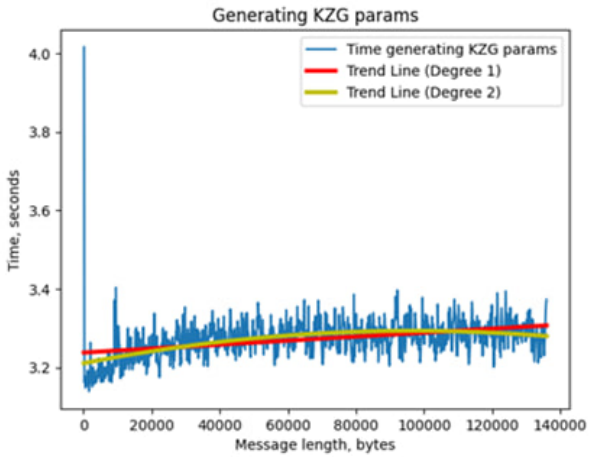


(d) Image 4: Keccak proof generation

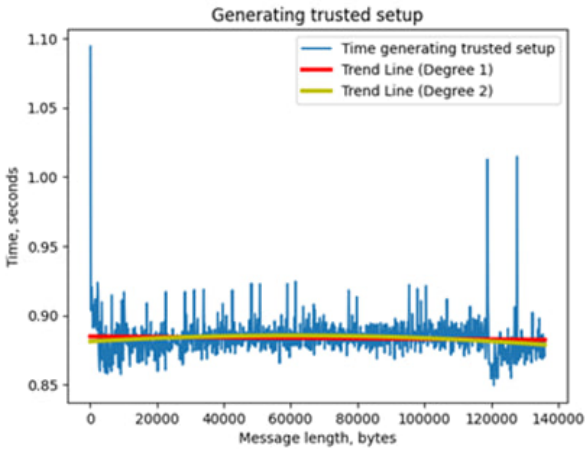


(e) Image 5: Proof verification

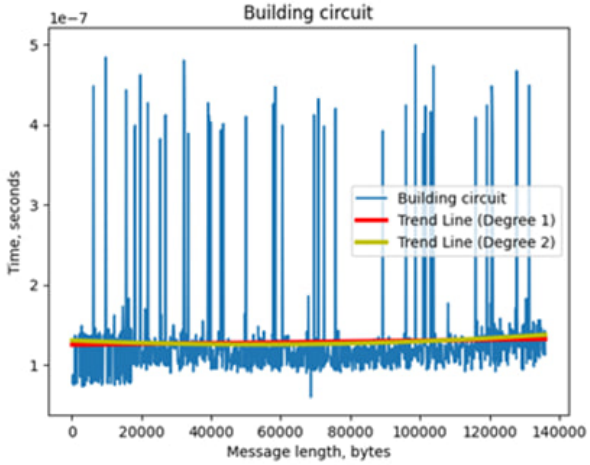
Figure 5: Axiom bench with params k=14, keccak_rows=25 on Linux



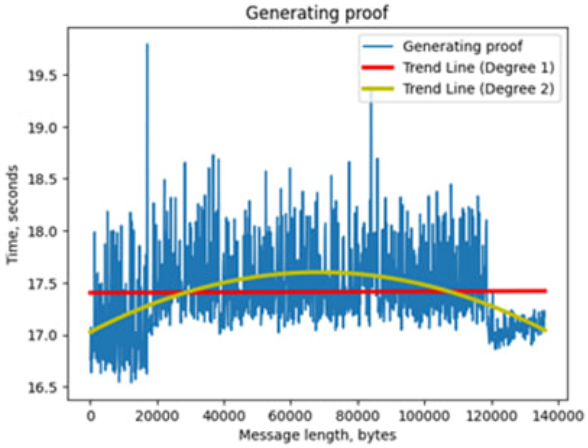
(a) Image 1: : KZG parameters generation



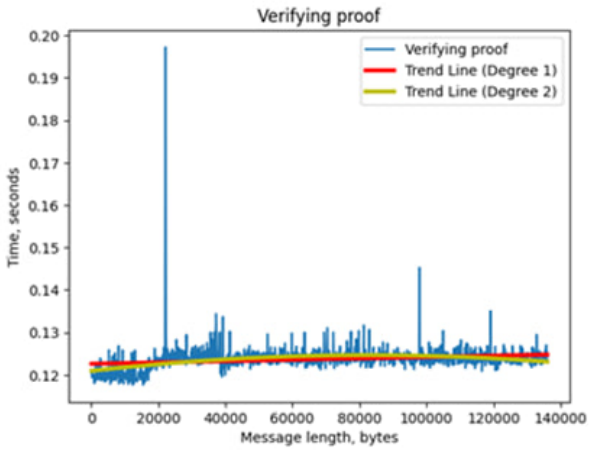
(b) Image 2: Trusted setup generation for circuit



(c) Image 3: Circuit building

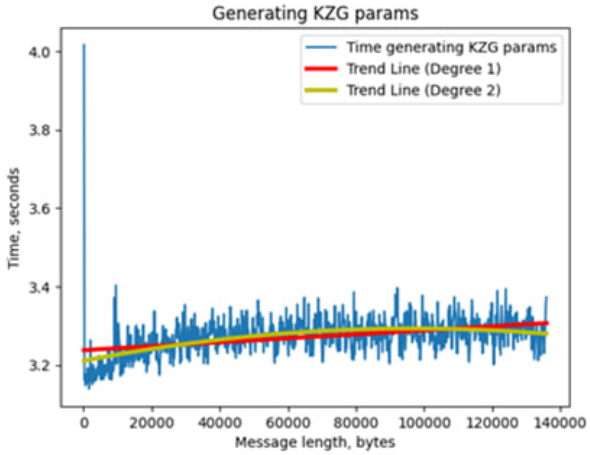


(d) Image 4: Keccak proof generation

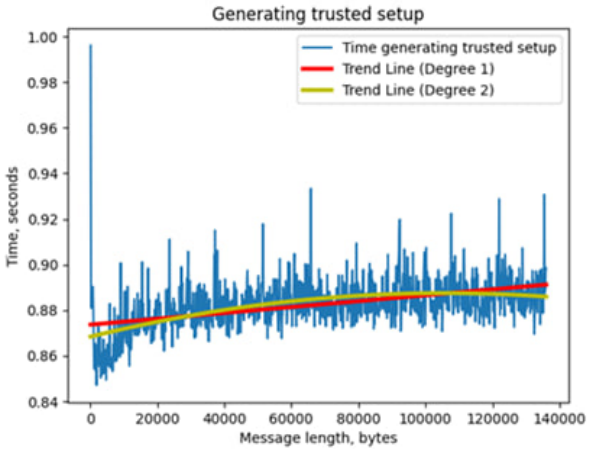


(e) Image 5: Proof verification

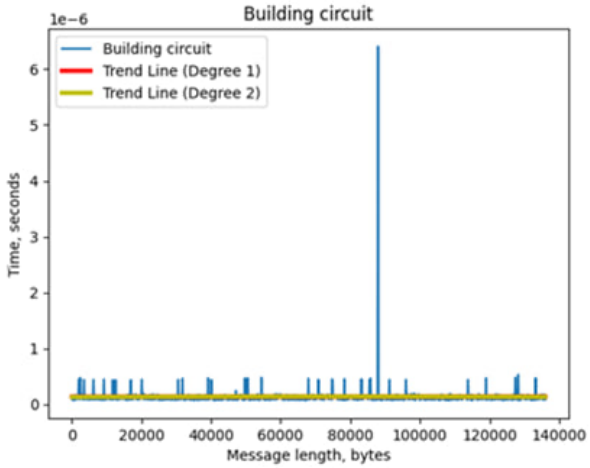
Figure 6: Axiom bench with params k=15, keccak_rows=28 on Linux



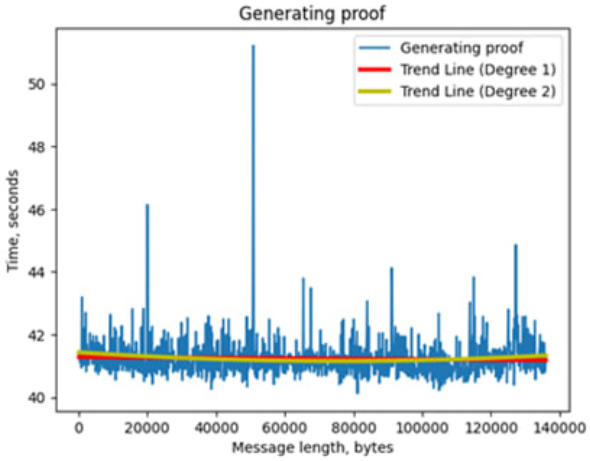
(a) Image 1: : KZG parameters generation



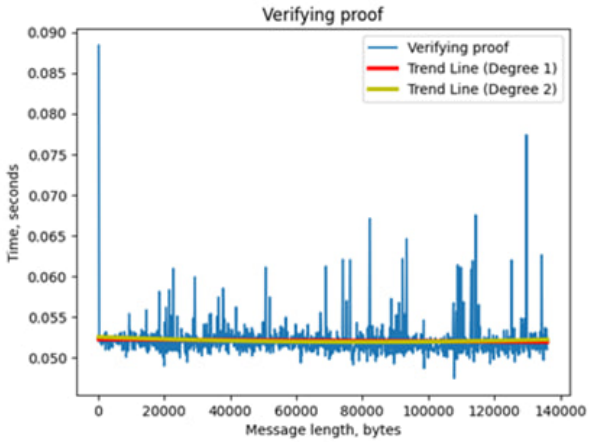
(b) Image 2: Trusted setup generation for circuit



(c) Image 3: Circuit building



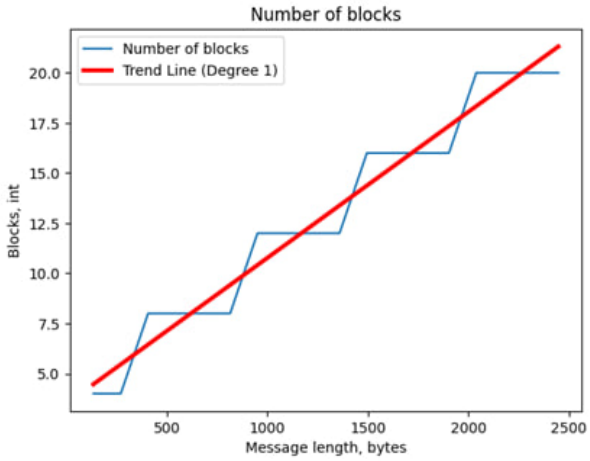
(d) Image 4: Keccak proof generation without including setup



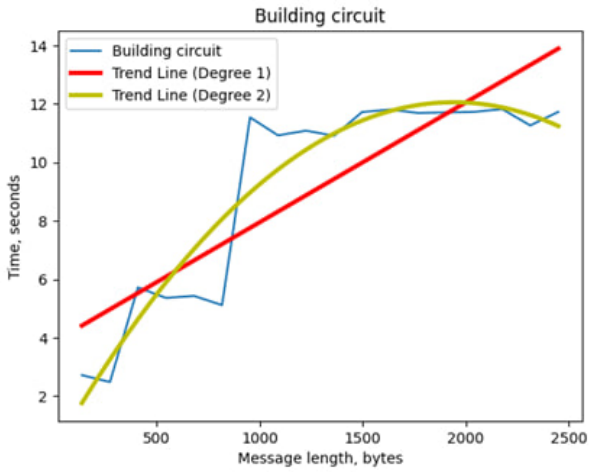
(e) Image 5: Proof verification

Figure 7: Axiom bench with params k=15, keccak_rows=9 on Linux

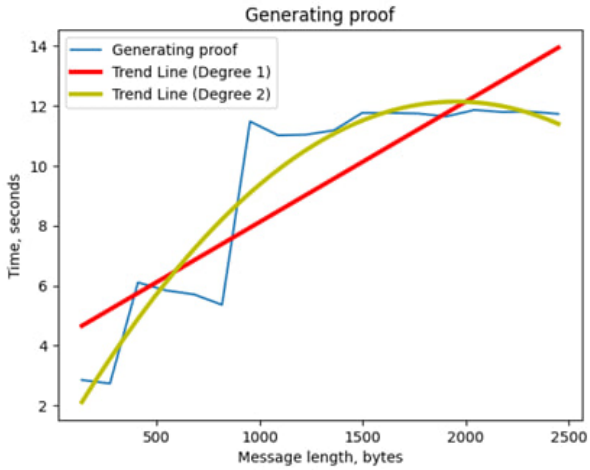
7.4 JumpCrypto implementation



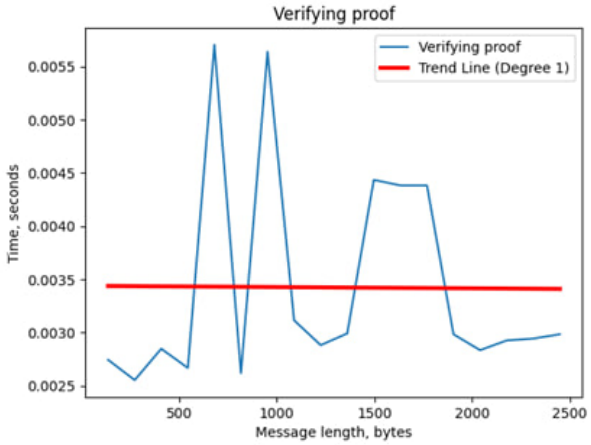
(a) Image 1: Increasing number of blocks depending



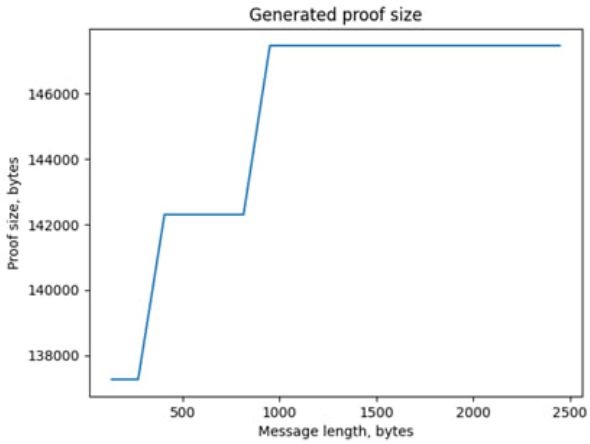
(b) Image 2: Circuit building



(c) Image 3: Keccak proof generation



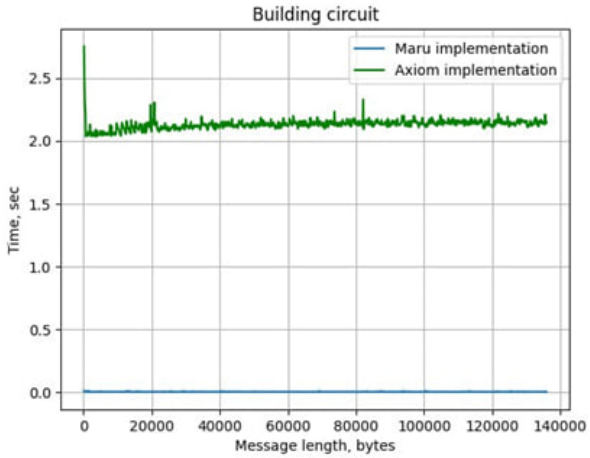
(d) Image 4: Proof verification



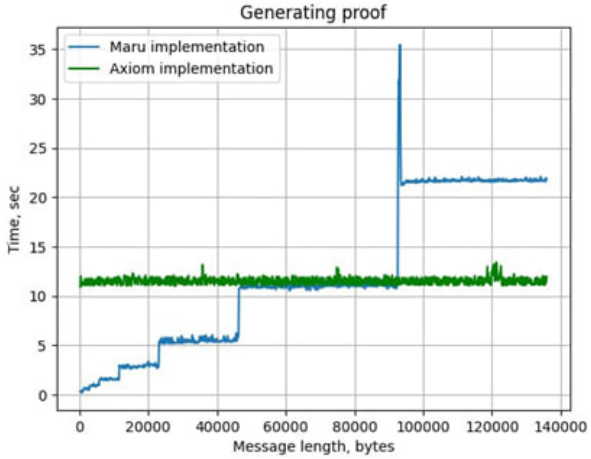
(e) Image 5: Proof size

Figure 8: JumpCrypto bench on Linux

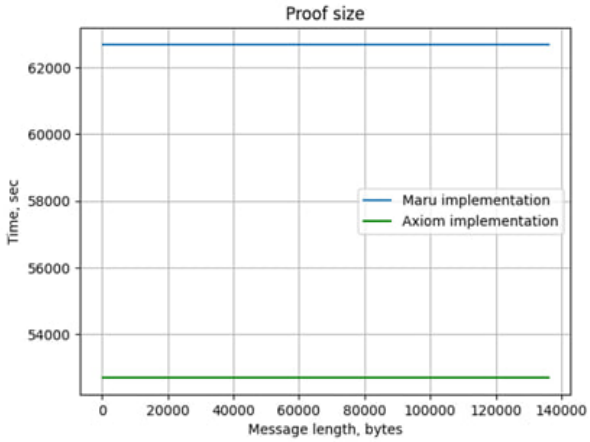
7.5 Comparison of Maru and Axiom implementations



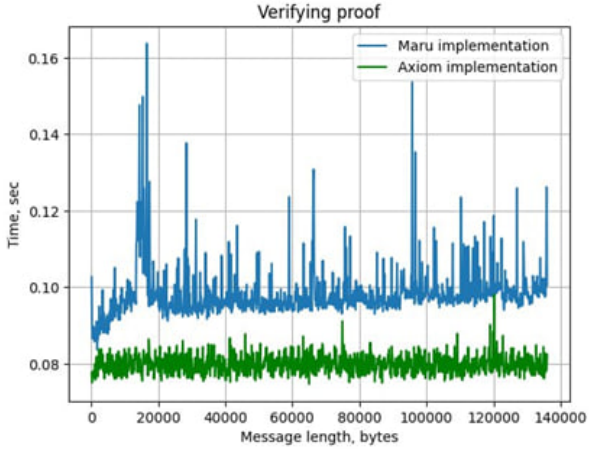
(a) Image 1: Increasing number of blocks depending



(b) Image 2: Proof generation



(c) Image 3: Proof size



(d) Image 4: Proof verification

Figure 9: Comparison of Maru and Axiom

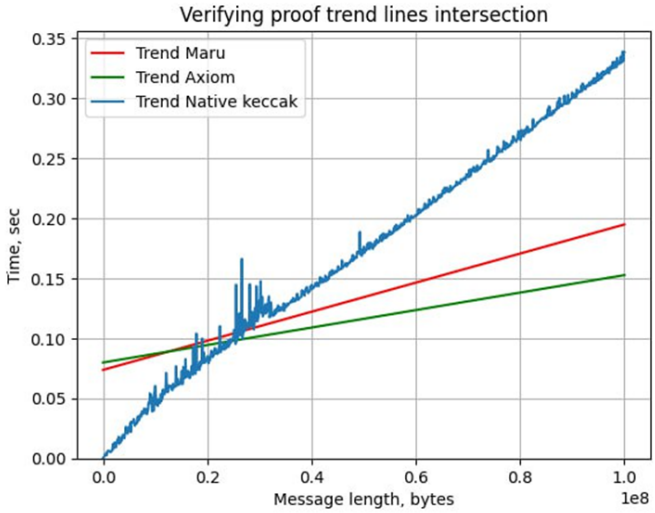


Figure 10: Effectiveness of implementations comparing with native keccak verification