Folding-Based Zero-Knowledge Proofs of Post-Quantum Signatures

Report by: Aryan Nath

Third-Year Computer Science Major and Mathematics Minor

Ashoka University, Haryana

Advisor: Prof. Saravanan Vijayakumaran

August 13, 2025

Abstract

This project proposes to design folding-based R1CS circuits which encode the verification algorithms of the post-quantum signature schemes ML-DSA and Falcon. We will also develop regular R1CS circuits for PQ signature verification which do not use folding, to quantify the improvements due to folding. Our end goal is to demonstrate that folding-based R1CS circuits for PQ signature verification can be proved and verified using significantly less computational resources compared to regular R1CS circuits. The tasks for this summer involved developing R1CS circuits for the SHA-3 family of hash functions which are derived from the Keccak algorithm, developing an Incrementally Verifiable Computation (IVC) Scheme for Keccak, using the IVC to create folding based proofs for the hash functions, and comparing the performance in creating the folding based proofs against the direct implementation.

Report

0.1 R1CS Circuits

To prove a statement we using zkSNARKS, we need to be able to represent that statement as an arithmetic circuit. One way of getting this representation is using Rank-1 Constraint Systems (R1CS).

Suppose the prover wants to prove to the verifier that it knows an x such that $y = x^3 + 6$. The prover makes y public but keeps x a secret. The prover generates the following constraints:

$$x_1 = (x) * (x)$$

$$x_2 = (x_1) * (x)$$

$$y = (x_2 + 6) \cdot (1)$$

Here $1, x, x_1, x_2, y$ would be a part of the witness vector known to the prover, only y would be known to the verifier, and the above constraint link x to y. So if the above statements are used to create a proof then the prover will be able to prove to the verifier in zero-knowledge that it possesses such an x.

An R1CS circuit collects all these statements into the following condensed form:

$$\left(u_{0,i} + \sum_{j=1}^{n} a_j u_{j,i}\right) \cdot \left(v_{0,i} + \sum_{j=1}^{n} a_j v_{j,i}\right) = \left(w_{0,i} + \sum_{j=1}^{n} a_j w_{j,i}\right)$$

where i is the index representing a statement, then this can represented as the following matrix multiplication:

$$\begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_{n'} \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \odot \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_{n'} \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \vdots \\ \mathbf{w}_{n'} \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

where $\mathbf{u_i}, \mathbf{v_i}, \mathbf{w_i}$ represent row vectors of n prime field elements (which can also be 0), and $\mathbf{a}$ is the witness vector.

Similar operations such as the following exist for boolean values:

1. Constrain a_1, a_2, a_3 as a boolean inputs

$$a_1(1 - a_1) = 0$$

$$a_2(1 - a_2) = 0$$

$$a_3(1 - a_3) = 0$$

2. AND Gate:

$$a_1 a_2 = a_3$$

   This means that if a_1, a_2, a_3 are boolean inputs, then if the constraint $a_1 a_2 = a_3$ does not hold then $a_3 = a_1 \wedge a_2$ is not true.

3. OR Gate:

$$(1 - a_1) \cdot (1 - a_2) = 1 - a_3$$

   This means that if a_1, a_2, a_3 are boolean inputs, then if the constraint $(1 - a_1) \cdot (1 - a_2) = 1 - a_3$ does not hold then $a_3 = a_1 \vee a_2$ is not true.

4. XOR Gate:

   $a_3 = a_2 \oplus a_2$ is represented as:

$$(a_1 + a_1) \cdot a_2 = a_1 + a_2 - a_3$$

5. NOT Gate:

   $a_2 = \neg a_1$ is represented as:

$$(1 - a_1) \cdot 1 = a_2$$

These constraints are used as building blocks for creating the R1CS circuits for complex algorithms like Keccak.

0.2 R1CS Circuits for Keccak

We have followed the FIPS 202 specification for SHA-3 family of hash functions: SHA3-256, SHAKE128, and SHAKE256.

$$\text{SHA3-256}(M) = \text{KECCAK}[512]\big(M \,\|\, 01,\, 256\big);$$

$$\text{SHAKE128}(M, d) = \text{KECCAK}[256]\big(M \,\|\, 1111,\, d\big),$$

$$\text{SHAKE256}(M, d) = \text{KECCAK}[512]\big(M \,\|\, 1111,\, d\big).$$

Figure 1 gives a brief overview of the Keccak algorithm, for more detail please refer to FIPS 202

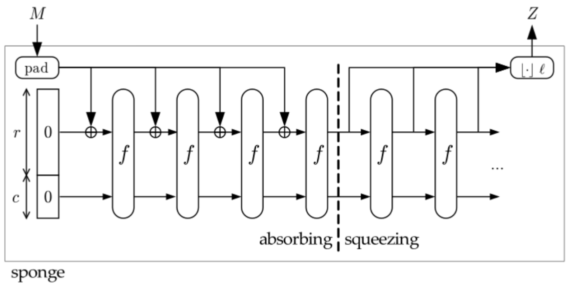
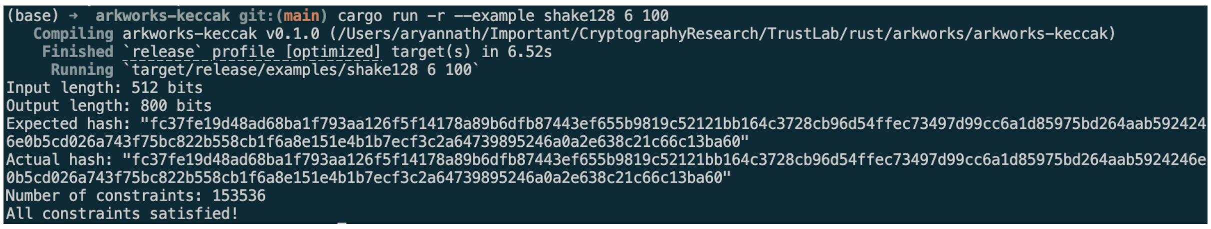


Figure 1: Sponge Construction for Keccak

f is permutation function which consists of operations using bitwise XOR, matrix column shifts, bitwise rotation, AND, and NOT on **UInt64** variables, and we constructed our own gadget to create constraints for the last two operations on UInt64 variables.

We used the **Arkworks Library** in **Rust** to create the R1CS circuits. The implementation can be found at `https://github.com/natharyan/arkworks-keccak`. Figure 2 is a snippet of the output:



```
(base) ➜  arkworks-keccak git:(main) cargo run -r --example shake128 6 100
   Compiling arkworks-keccak v0.1.0 (/Users/aryannath/Important/CryptographyResearch/TrustLab/rust/arkworks/arkworks-keccak)
    Finished `release` profile [optimized] target(s) in 6.52s
     Running `target/release/examples/shake128 6 100`
Input length: 512 bits
Output length: 800 bits
Expected hash: "fc37fe19d48ad68ba1f793aa126f5f14178a89b6dfb87443ef655b9819c52121bb164c3728cb96d54ffec73497d99cc6a1d85975bd264aab592424
6e0b5cd026a743f75bc822b558cb1f6a8e151e4b1b7ecf3c2a64739895246a0a2e638c21c66c13ba60"
Actual hash: "fc37fe19d48ad68ba1f793aa126f5f14178a89b6dfb87443ef655b9819c52121bb164c3728cb96d54ffec73497d99cc6a1d85975bd264aab5924246e
0b5cd026a743f75bc822b558cb1f6a8e151e4b1b7ecf3c2a64739895246a0a2e638c21c66c13ba60"
Number of constraints: 153536
All constraints satisfied!
```

Figure 2: A snippet of the output of our implementation of an R1CS circuit for Keccak.

0.3 Incrementally Verifiable Computation

In order to create a folding proof, we need to be able to represent an algorithm in the form of an *incrementally verifiable computation* scheme. A *uniform* incrementally verifiable computation scheme allows a prover to prove to the verifier that for some public step function F, public initial input z_0, public final output z_n, it knows auxiliary input values $w_0, w_1, \dots, w_{n-1} \in \mathbb{F}^{k-1}$ such that

$$z_n = F\big(F(\dots F(F(F(z_0, w_0), w_1), w_2) \cdots, w_{n-2}), w_{n-1}\big)$$

where $z_{i+1} = F(z_i, w_i)$ for each $i \in \{0, 1, \dots, n-1\}$, and z_i and z_{i+1} are the public input and output in the ith step, respectively.

Signature verification in both ML-DSA and Falcon involves the repeated invocation of the Keccak hash function (specifically, SHAKE128 and SHAKE256 are invoked 256 to 512 times). Each invocation of Keccak256 hash function requires approximately **153,536** R1CS constraints. For context, in total we would require **39 million** R1CS constraints just to constraint the Keccak computations. By using a folding scheme, we aim to reduce the huge R1CS matrix to a much smaller (base computation) matrix. That, is if there are $\mathbf{T}$ invocations of a base computation (step function) with $\mathbf{C}$ constraints, then in a direct implementation the total R1CS constraints would be approximately $\mathbf{C} \times \mathbf{T}$; however, using a folding proof the number of R1CS constraints in the final circuit remain $\mathbf{C}$.

We constructed the incrementally verifiable computation for Keccak as follows:

Algorithm 1 Step function algorithm for $z_i = F(z_{i-1}, w_{i-1})$ in step i

Input:
- If $i > 1$, public input $z_{i-1} = [\texttt{cur\_opcode}, \texttt{cur\_hash}_{i-1}]$
- If $i = 1$, then $z_{i-1} = z_0 = [0, \mathbf{0}]$
  (opcode contains first `n_steps` bits as the step index and the most significant bit as the hash flag and the initialization string is a 0-bit string)

Auxiliary input: $w_{i-1} = [\texttt{next\_opcode}, \texttt{n\_steps}, M_i, \texttt{cur\_digest}]$
(`n_steps` is the total iterations required over the spong construction and M_i denotes the message block input to the current step)

Output: Public output $z_i = [\texttt{next\_opcode}, \texttt{next\_hash}]$.

```
// Check that the cur_opcode and next_opcode are of (n_steps+1) bits
left_is_smaller(cur_opcode, pow2(n_steps+1))
left_is_smaller(next_opcode, pow2(n_steps+1))
// Decompose the current and next opcodes into the constituent flag bit and step index
$(cur\_flag, cur\_index)$ ← decompose_opcode(cur_opcode)
$(next\_flag, next\_index)$ ← decompose_opcode(next_opcode)
// Next step index should always be 1 more than the current index
assert(next_index == cur_index + 1)
// If current flag is 0, next flag can be 0 or 1; if current flag is 1, next flag must be 1 (can switch from absorption to squeezing phase, but not the other way round
assert(cur_flag $\wedge \neg$ next_flag == 0)
// In the squeezing phase, $M_i$ must be a zero string
assert($\neg$cur_flag $\vee$ ($M_i$ == $\mathbf{0}$))
// Create flags
is_first_step ← create_flag(cur_index == 0)
// Check that the non-deterministic inputs hash to the expected value
expected_cur_hash ← $f(M_i, cur\_digest, cur\_flag)$
// Equality check skipped for first step
assert(is_first_step $\vee$ (expected_cur_hash == cur_hash))
// Compute the next Keccak digest using $f$ (cond_select returns 2nd arg if true, else 3rd)
next_hash ← $f(M_i, cur\_digest, cur\_flag)$
// No nullifier required for verifying a single-run of Keccak
$z_i$ ← [next_opcode, next_hash]
```

0.4 Folding Based Proof for Keccak

We have used LatticeFold (Boneh and Chen) for our folding scheme. In addition to reducing the size of the R1CS matrix, LatticeFold uses post-quantum secure commitments which keep the prover's witnesses secure against *harvest now, decrypt later* attacks.

Using this scheme we were able to bring down the number of R1CS constrants for a single invocation of Keccak hash function from **153,536** to **37,000**. Our implementation can be found at https://github.com/natharyan/latticefold-keccak.

Acknowledgement

I would like to thank Prof. Saravanan Vijayakumaran for his continuous support and guidance throughout the project, and for giving me the opportunity to work with him. I would also like to thank the IITB Trust Lab Team for giving me this opportunity and for helping to make the onboarding process smooth!