ZK friendly hashes

Karthik Inbasekar

Ingonyama

20 Oct 2022

Introduction

Cryptographic hash functions

A cryptographic hash function is a map that projects a value from a larger set (message) to a value in a smaller set (digest)

D: Digest $\in \{0,1\}^k \leftarrow y = H(X)$ **Message** $X \in \{0,1\}^j$

$|k| < |j|$

One way: Pre-image resistance. Given y it is computationally infeasible to reconstruct X

Collision resistance:

Weak: Given a $X \in M$, It is computationally infeasible to find a $X' \in M$ such that $H(X) = H(X')$

Strong: It is computationally infeasible to determine any $X \neq X' \in M$ such that $H(X) = H(X')$

Security parameter

If the hash is a truly random map to $\{0,1\}^k$ then an attacker has to evaluate it $\sim 2^{k/2}$ times to find a collision

Birthday paradox

Zero Knowledge Proofs (ZKP's) and hashes

A ZKP is a protocol between two entities: Prover and Verifier, that allows a prover to convince the verifier about the truth of a statement, without revealing any additional information.

NP relation

$$y = H(x, w)$$

x : public

w: private

Completeness: If the NP relation is true: A prover that follows the prescribed protocol (honest) will be able to convince a honest verifier that relation holds with probability 1.

Soundness: If the NP relation is false: The probability that a malicious prover will be able to convince a honest prover that the relation holds is negligible.

Zero Knowledge: If the NP relation is true: a malicious verifier does not learn anything about w

Zero Knowledge Proofs (ZKP's) and hashes

The one way and collision resistant properties of hashes make them ideal candidates for expressing as a NP relation

The NP relation is expressed as an **arithmetic circuit** that consists of gates and wires, In general **wire values in ZK are elements in a prime field**

Large bit sized numbers

Traditional time tested hash functions like SHA256 possess **enormous amounts of bitwise operations** (CPU friendly)

Each bitwise XOR or AND operation would need one multiplication or one addition gate

This leads to

Large polynomial degrees

Large Circuit sizes

Increased FFT complexity

Increased prover time

Look up tables can change this scenario

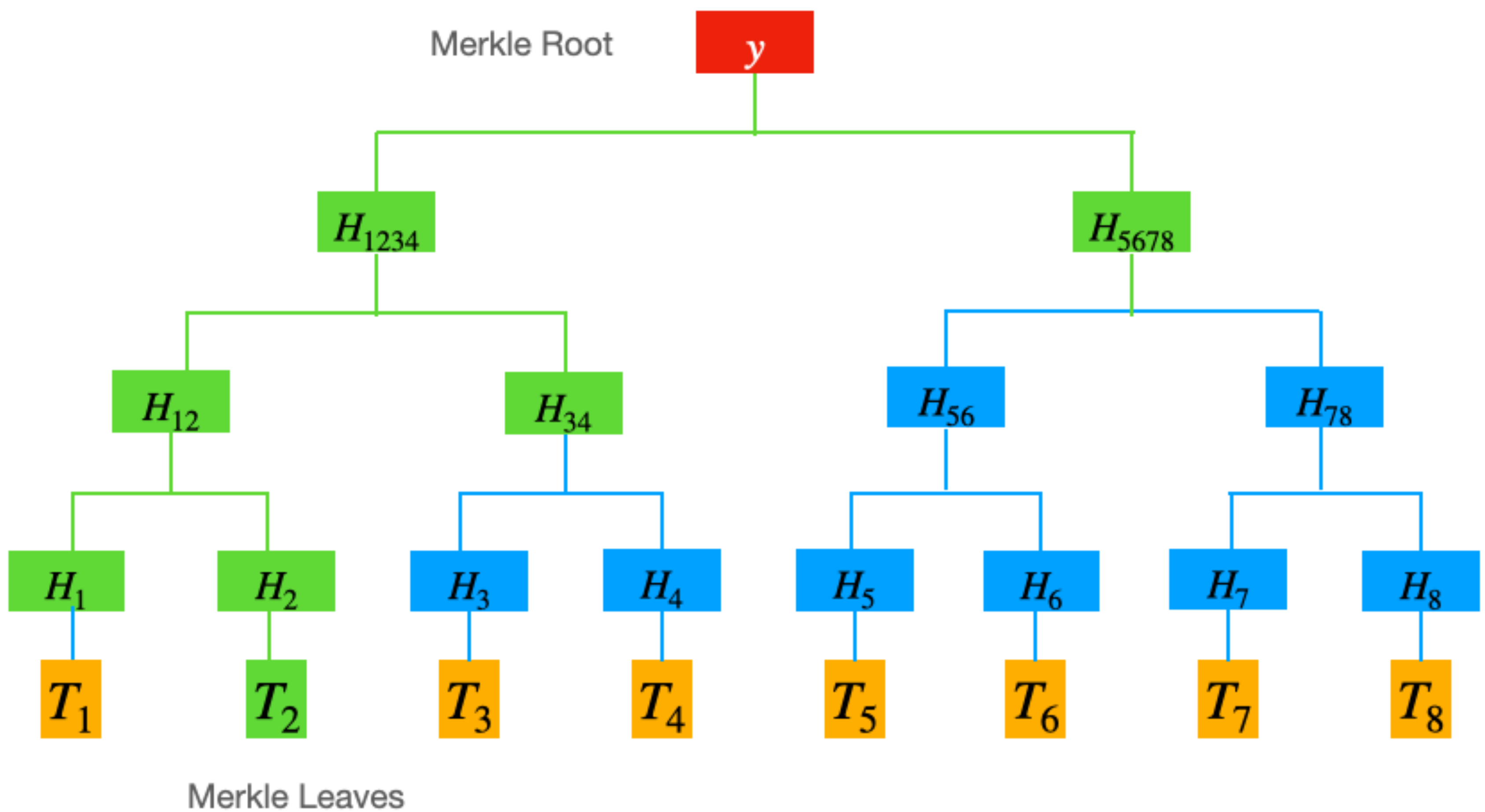
Usage of hashes in ZKP : NP relation

As a NP relation, such as a sequence of hash functions

$$y = H(\ldots H(H(H(w_0, w_1), w_2), w_3) \ldots, w_{r-1})$$

w_i : private , H, y are known and public

Eg : Proof of membership of a leaf in a Merkle tree



Usage of hashes in ZKP : NP relation

Finite field friendly **Natively defined on finite fields**

The time of execution of the hash is less important than the arithmetization of the hash. **Arithmetization oriented hashes**

Minimize arithmetic complexity to obtain low degree polynomials and constraint relations **Hash must contain simple addition/ multiplication/power operations**

Security **As required by the application**

Bonus: **Collision resistance properties of hashes are considered to be safe against Quantum attacks**

Usage of hashes in ZKP: Commitments and Fiat-Shamir

Hash functions as cryptographic commitments

Each leaf in the Merkle tree can be an element of a vector/ coefficient of a polynomial to be committed to.

Sequence of hashes, with the root being commitment to original data

Time of execution of the hashes should be optimal

Fiat-Shamir transforms

Fiat-Shamir transforms convert a interactive protocol into a non-interactive one with an agreed upon random oracle.

The output of a hash function is equivalent to a pseudo random field element.

Not a bottleneck

What characteristics do we want for ZK friendly hashes?

Natively defined to operate on finite field elements

Low arithmetic complexity (constraints/polynomial degrees)

Arithmetic operations include: addition/multiplication operations and potentially lookups

Fast execution times for commitments

If application requires b bits security, the hash must have at least b bits security

Usual properties: **Oneway and Collision resistance**

SPN (Substitution and Permutation Networks) capture much of the above

Battle testing: Security audits that give ZK friendly hashes the durability status

SPN's based Sponge construction and design

Framework: Sponge Construction

SPN: Arbitrary input/output sizes, but fixed length permutations

Permutations:

Linear layer : Element wise additions and Matrix multiplications

Spread the elements uniformly in the field

Non-Linear layer : Power maps

Increase the degree of the permutation

Parameters

p Prime modulus

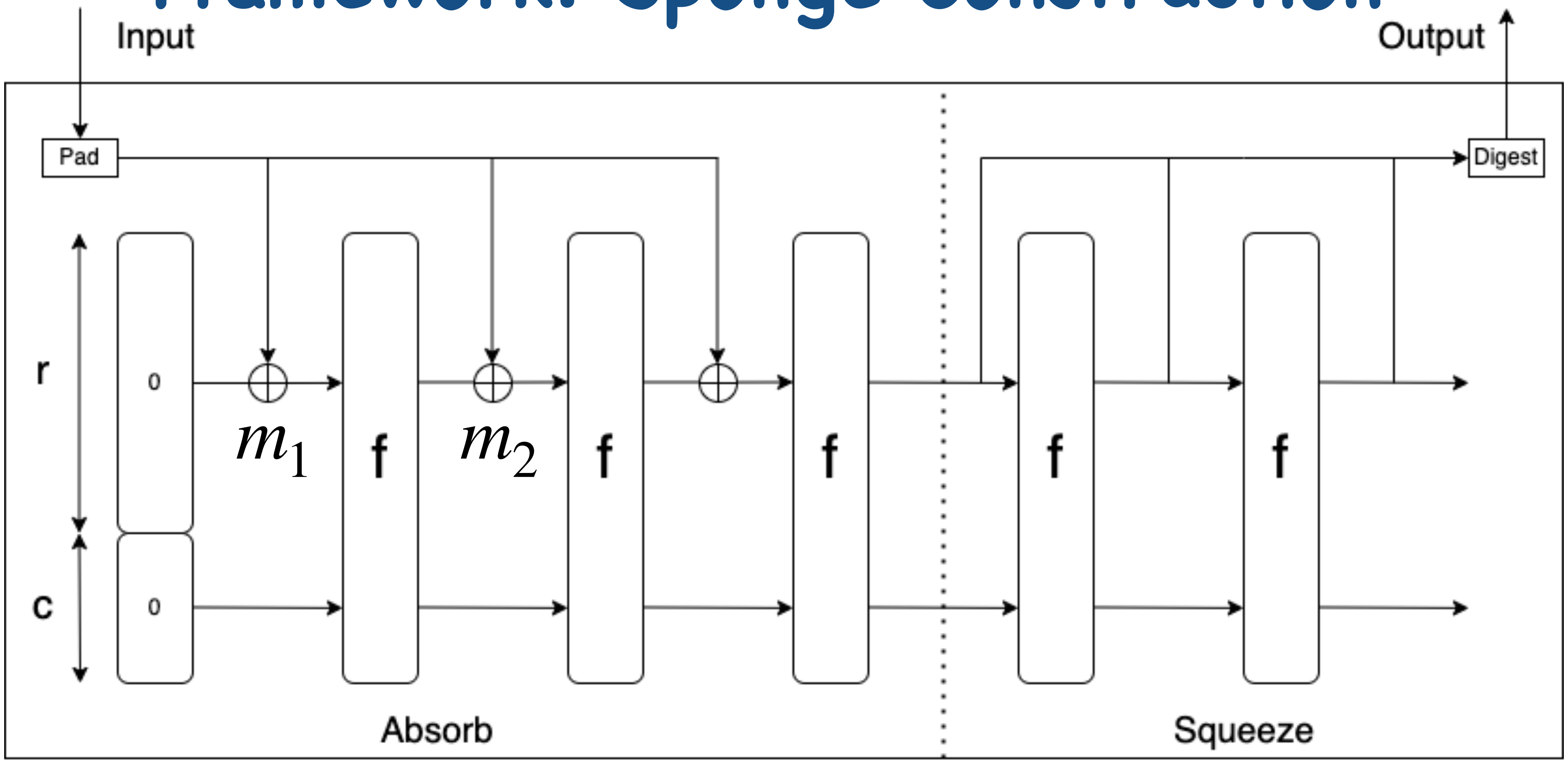
t State size $\texttt{state}[i] \in \mathbb{F}^{r+c} \quad \forall i \in \{0,1,\ldots t-1\}$

S Security in bits $S = \log_2(\sqrt{p}) \cdot \min(r, c)$

r: digest size $t = \text{len(input)} + \left\lceil \frac{2S}{\log_2(p)} \right\rceil$

Eg: in a 256 bit prime field, a 128 bit secure hash has r=1

Framework: Sponge Construction



The sponge permutation takes a input of size t, applies a series of fixed length $|t|$ permutations and outputs a digest of r elements

In general if message: M is long, $M : [m_1||m_2||\ldots]$; $|m_i| = r$

Absorb $$f([\ldots f([m_{i+1} \oplus f([m_i \oplus r_i||c_i])||c_{i+1}])\ldots])$$

Goal is to have as many rounds of f to reach uniform distribution in the state

Squeeze Output final digest

Sponge Construction : general structure of round functions

In general the round function consists of the following building blocks

Non-Linear layer : Power maps

$\pi_0 \colon x \to x^{\alpha}$ $\quad\quad\quad \alpha \geq 3$

$\pi_1 \colon x \to x^{-1}$ $\quad\quad\quad \mathrm{gcd}(\alpha, p-1) = 1$

invertibility and provides non-linearity

Depending on the design of the hash sometimes either or both are used

The main reason to include the power map, is to increase the degree of the elements such that the security requirements are met.

Linear layer

$\mathrm{state} = \mathrm{state} \overrightarrow{\oplus} \mathrm{RoundConstants}_j$ — Element wise field additions

$\mathrm{state} = \mathrm{state} \times \mathscr{M}_{t \times t}$ — MDS Matrix multiplication

Round constants: eg like Public Keys,
MDS (Maximum Distance Separable) matrix: is to spread the uniform randomness properties across the entire state.

Sponge Construction : general structure of round functions

The round constants and MDS matrices are recomputable given the triple (t, p, S)

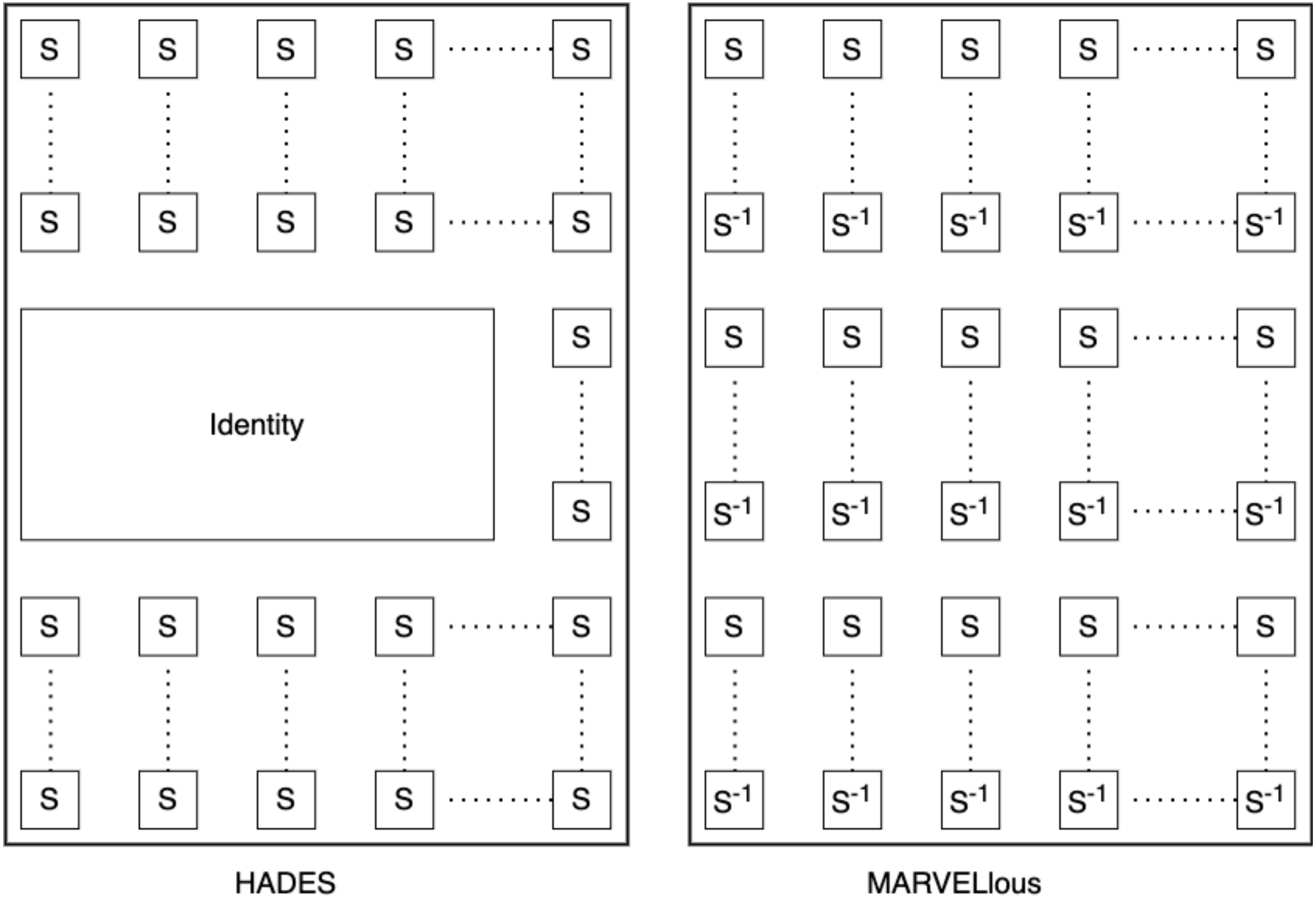
The round constants for instance are computable using a cipher: Grain LFSR (Linear Feedback Shift Register)

A simple Algorithm to compute a MDS matrix

1. Find $g \in \mathbb{F}_p$, the smallest primitive root
2. Build Vandermonde matrix $V \in \mathbb{F}_p^{t \times 2t}$, $V[i, j] = g^{ij}$, $i \in 0, \ldots t$, $j \in 0, \ldots 2t$
3. Bring V to reduced row echelon form
4. $V \equiv [I \,|\, M^T]$

Rescue paper

Sponge Construction : Design



Low computational complexity

High computational complexity

It is all relative!

Less secure

More Secure

Sponge Construction : Design

HADES design

S-boxes are unevenly distributed across several rounds.

Low computational complexity

Partial rounds: S-box is only applied to some of the elements in a state

Full rounds: S-box is applied to all elements in a state.

Security: Partial rounds are always sandwiched in one batch in-between two batches of full rounds. The Full rounds protect against statistical attacks and the internal rounds raise the degree of the permutations.

Eg: Poseidon/Reinforced concrete

MARVELlous design

S-boxes are evenly distributed across several rounds.

More computational complexity:

Use both power maps : π_1 in even rounds and π_2 in odd rounds.

π_2 is an inverse map, if π_1 is of low degree then π_2 is in general a very high degree map (and vice-versa) due to modular inversion.

More Secure

Eg: Rescue/Rescue-prime

Hades: Poseidon application - Filecoin

Filecoin

Overview:

Deals with (nodes) clients: to store and retrieve data

- Nodes: sync the block chain and validate messages in every block

Contribute **storage capacity**

- Decentralized p2p: several miners can store same set of data

Cryptographic proofs to verify storage across time

- Prove: I have stored all data submitted by client
- Prove: I have stored all data during the lifetime of the deal

Incentives:

Storage Fees: Paid by client for data storage and retrieval

Block rewards: FIL tokens for advancing the blockchain

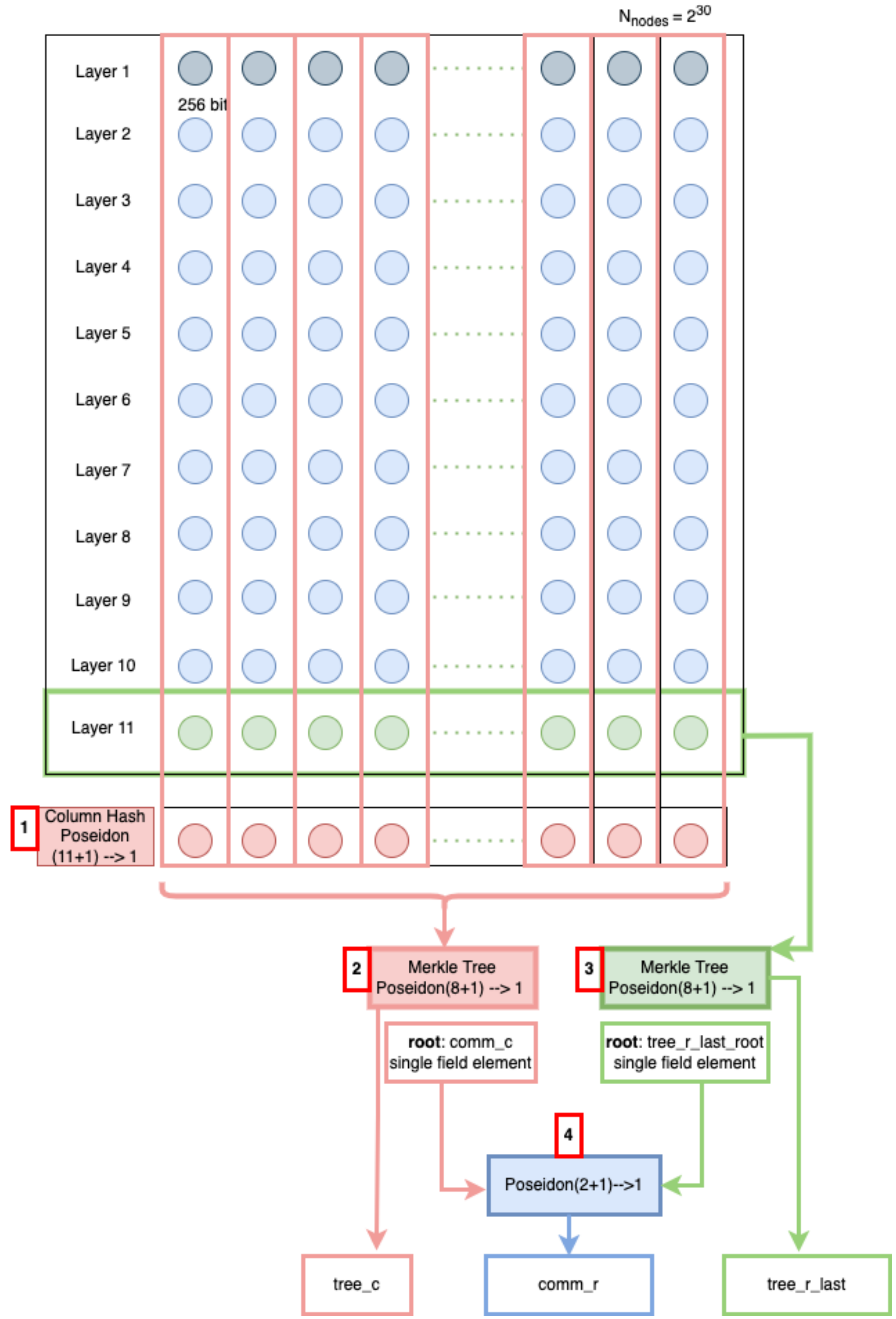
More Storage = More rewards

Filecoin

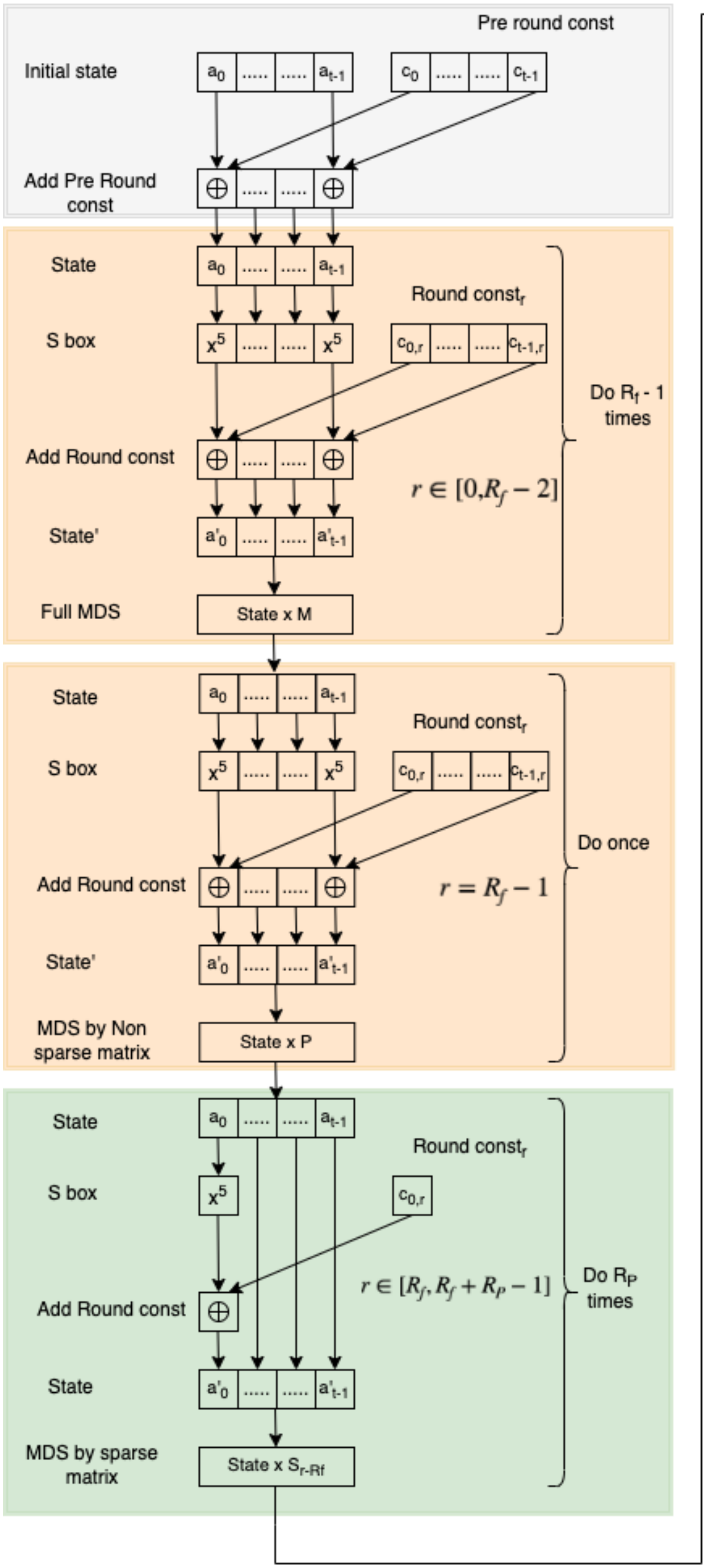
The files to be stored in filecoin are converted into a data structure (Replica) of an array of 11×2^{30} 256 bit field elements

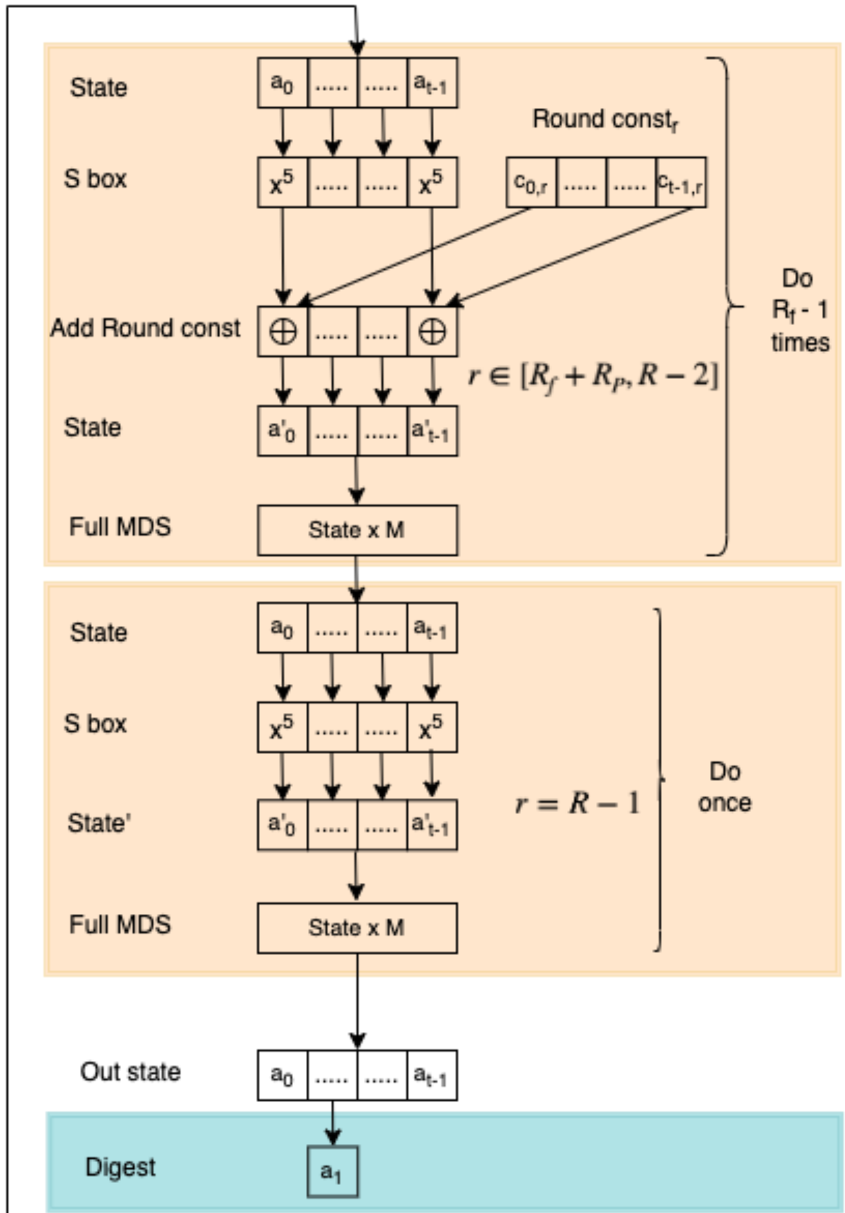
Multiple Poseidon Instances run in order to complete a process known as sealing that creates a copy of the data in local storage.

The end product of the instances are Merkle roots, that are used as cryptographic commitments to verify file storage.



Optimized Poseidon





256 bit field elements in scalar field of BLS12-381

Red: Full rounds

Green: Partial rounds

Digest size : 1

128 bit security

$$
\text{state} = \begin{cases} \text{state}[i]^5 \Big|_{i=0,1,\ldots,t-1} & \text{Full round} \\ \text{state}[0]^5 & \text{Partial round} \end{cases}
$$

Optimization:

In partial rounds the linear layers are combined to effectively make the MDS matrix sparse.

Poseidon in Filecoin : Algebraic complexity

Poseidon instances

$\text{Poseidon}_{11}(\mathbb{F}_p^{[12]}) \to \mathbb{F}_p^{[1]}$ for each column in the array

Two $\text{Poseidon}_8(\mathbb{F}_p^{[9]}) \to \mathbb{F}_p^{[1]}$ Octinary Merkle tree (depth 10) instances

One $\text{Poseidon}_2(\mathbb{F}_p^{[3]}) \to \mathbb{F}_p^{[1]}$ Binary Merkle tree

Total hashes: $2^{30} + (2^{30} - 1)/7 * 2 + 1$ **7-8 min in RTX3090 GPU**

Eg: Lowering algebraic complexity is important for performance

Instantiation	t	input	(R_F, R_P)	Arity	A	M
$\texttt{Poseidon}_{11}(\mathbb{F}_p^{[12]}) \to \mathbb{F}_p^{[1]}$	12	$[2^{11}-1, t_1, t_2, \dots t_{11}]$	$(8, 57)$	-	2463	2922
$\texttt{Poseidon}_8(\mathbb{F}_p^{[9]}) \to \mathbb{F}_p^{[1]}$	9	$[2^8-1, t_1, t_2, \dots t_8]$	$(8, 56)$	8	1617	2004
$\texttt{Poseidon}_3(\mathbb{F}_p^{[3]}) \to \mathbb{F}_p^{[1]}$	3	$[2^2-1, t_1, t_2]$	$(8, 56)$	-	352	592

Table 2. Poseidon Instantiations in Filecoin: In the table A refers to the number of modular additions and M refers to the number of modular multiplications per hash invocation.

Poseidon in Filecoin : Arithmetization complexity

Proof of integrity of computation: Random nodes are selected in the beginning of the data structure and the prover is required to provide a valid path up to the public commitment (root)

Arithmetization: R1CS (Rank One Constraint System), constraints of a single instance

Instantiation	Proof of	R1CS constraints
$\texttt{Poseidon}_3(\mathbb{F}_p^{[3]}) \to \mathbb{F}_p^{[1]}$	Preimage	240
$\texttt{Poseidon}_8(\mathbb{F}_p^{[9]}) \to \mathbb{F}_p^{[1]}$	Merkle Tree	3416
$\texttt{Poseidon}_{11}(\mathbb{F}_p^{[12]}) \to \mathbb{F}_p^{[1]}$	Preimage	459

Table 3. Constraint sizes in R1CS for Filecoin Poseidon applications (128 bit security) evaluated at a vector size 2^{24} [27]

Polynomial sizes are of the order 2^{26}, and there is significant computation complexity in FFT and MSM (Multi Scalar Multiplication) in the Groth16 Proof

7 min in RTX3090 GPU

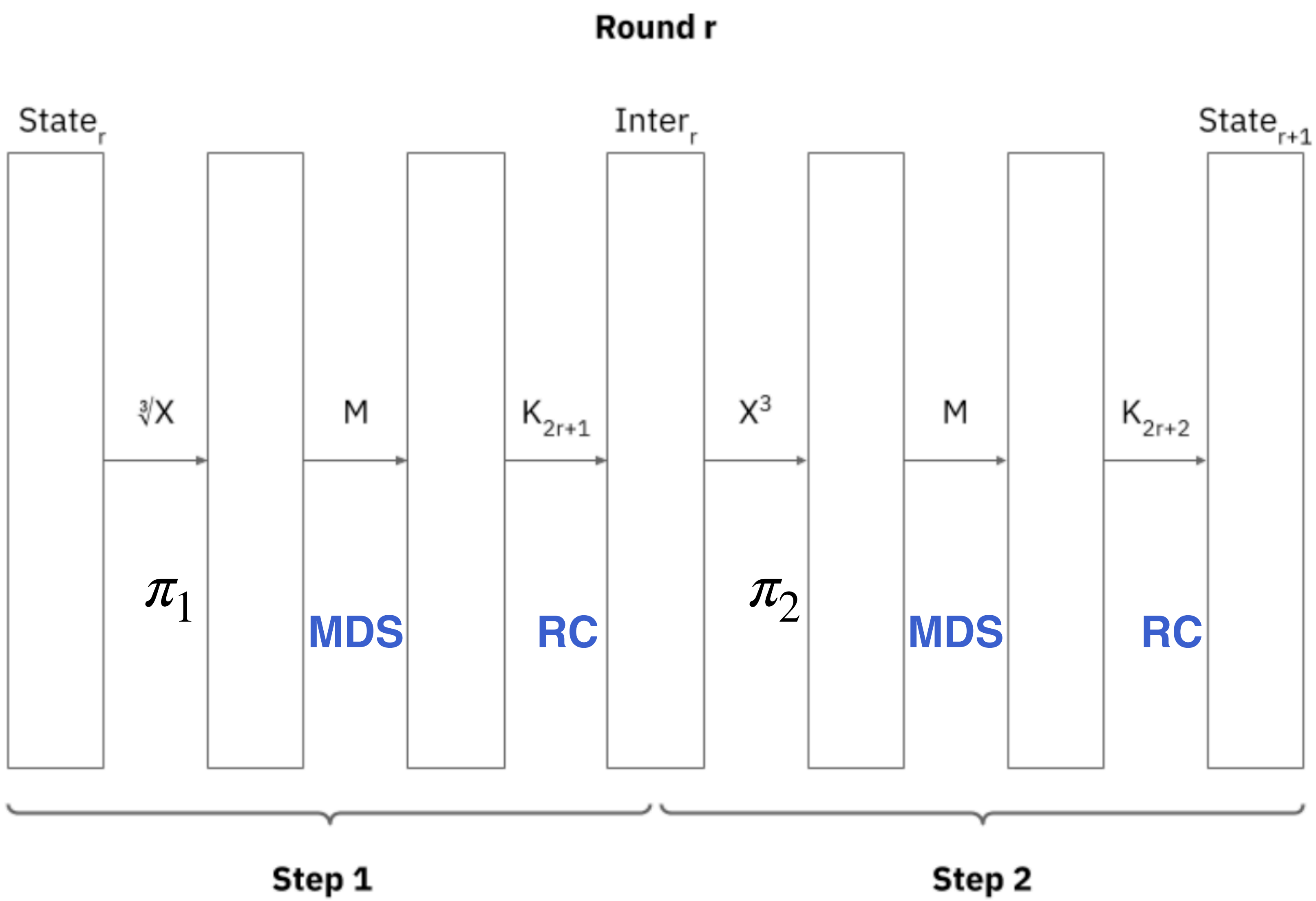
MARVELlous: Rescue application - EthSTARK

Invocation of Rescue Hash: Overview

Field $\mathbb{F}_p$ with $p = 2^{61} + 20 * 2^{32} + 1 = 2305843095113039873$

Inputs: $w = (w_0, w_1, \ldots w_n)$, where $w_i \in \mathbb{F}_p^4$

Rounds = 10 (determined by security level, in this case: 128 bits)



digest size $= \left\lceil \frac{2S}{\log_2(p)} \right\rceil = 4$

A bit about the S box

S - Box is an element wise exponentiation of the input w_i

$$\pi_1 : x^{1/\alpha} \qquad \pi_2 : x^{\alpha} \qquad \forall x \in \mathbb{F}_p$$

Where $\alpha \nmid p - 1$. For ethSTARK the value of $\alpha = 3$

Since $3 \nmid 2^{61} + 20 * 2^{32}$, it can be checked that $\dfrac{2p-1}{3} \in \mathbb{Z}$

$$\pi_1 : x^{1/3} \equiv x^{\frac{2p-1}{3}}$$

Cube root permutation in $\mathbb{F}_p$

$$\pi_2 : x^3$$

Cube permutation in $\mathbb{F}_p$

ethSTARK proof Statement

Prover: Knowledge of a sequence of inputs

$$w = \{w_0, w_1, \ldots, w_n\}$$

Such that

$$H(\ldots H(H(w_0, w_1), w_2), \ldots, w_n) = \text{output}$$

w_i are Field elements $\in \mathbb{F}_p$

The Chain is computed as

$$\begin{array}{llll}
H(w_0, w_1) \to o_1 & & & \\
\quad\downarrow & & & \\
\quad H(o_1, w_2) \to o_2 & & & \\
\qquad\qquad\downarrow & & & \\
& \ldots\ldots & & \\
& & \downarrow & \\
& & H(o_{n-1}, w_n) \to \text{output} &
\end{array}$$

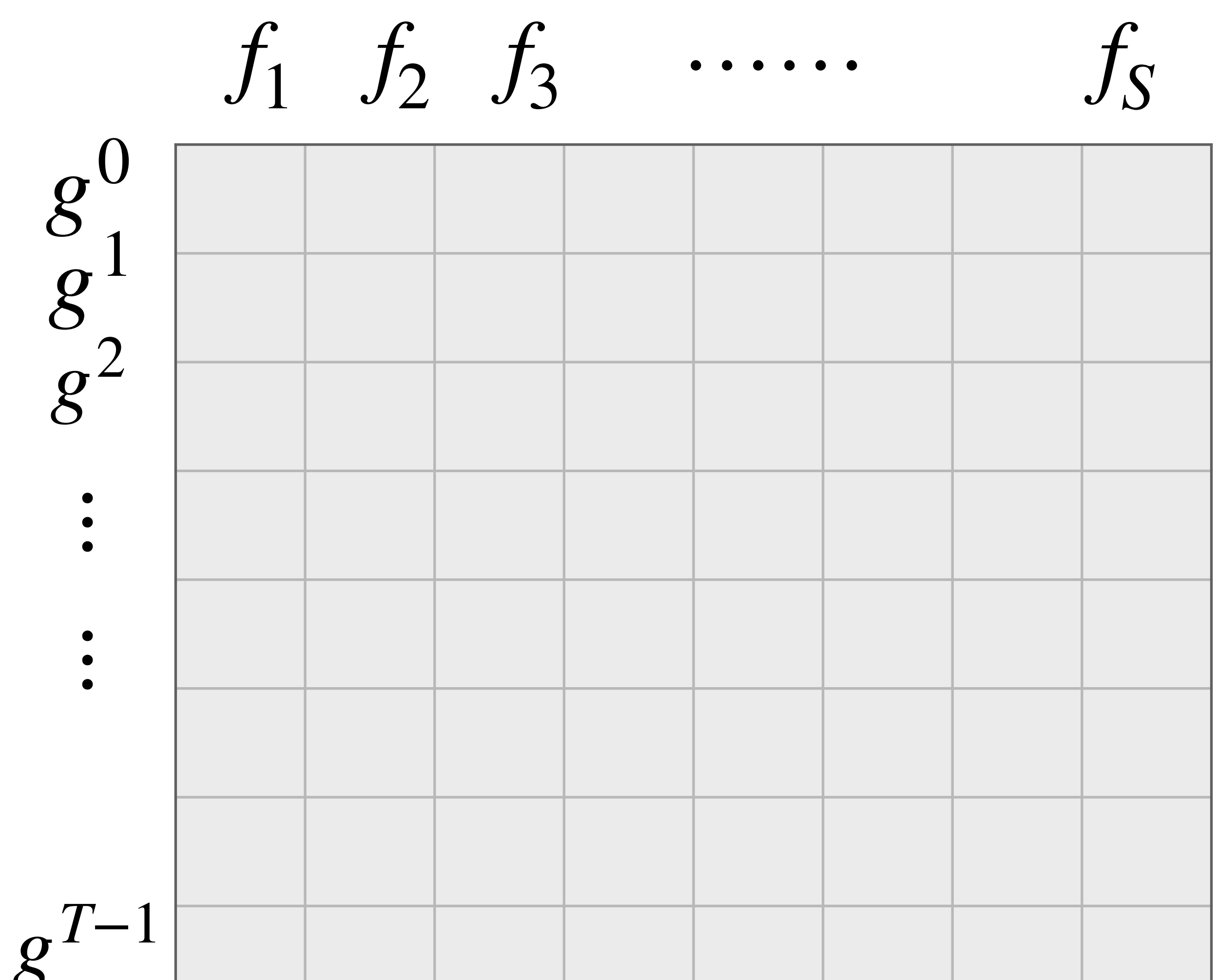
Output is public, $n = |w| - 1$ is the Hash chain length known to the verifier.

AIR: Execution Trace

An execution trace is a sequence of Machine states, one per clock cycle.

Computation: S registers and T cycles. Trace table: $S \times T$

Given the hash chain statement, the prover runs the arithmetic and builds an execution trace.



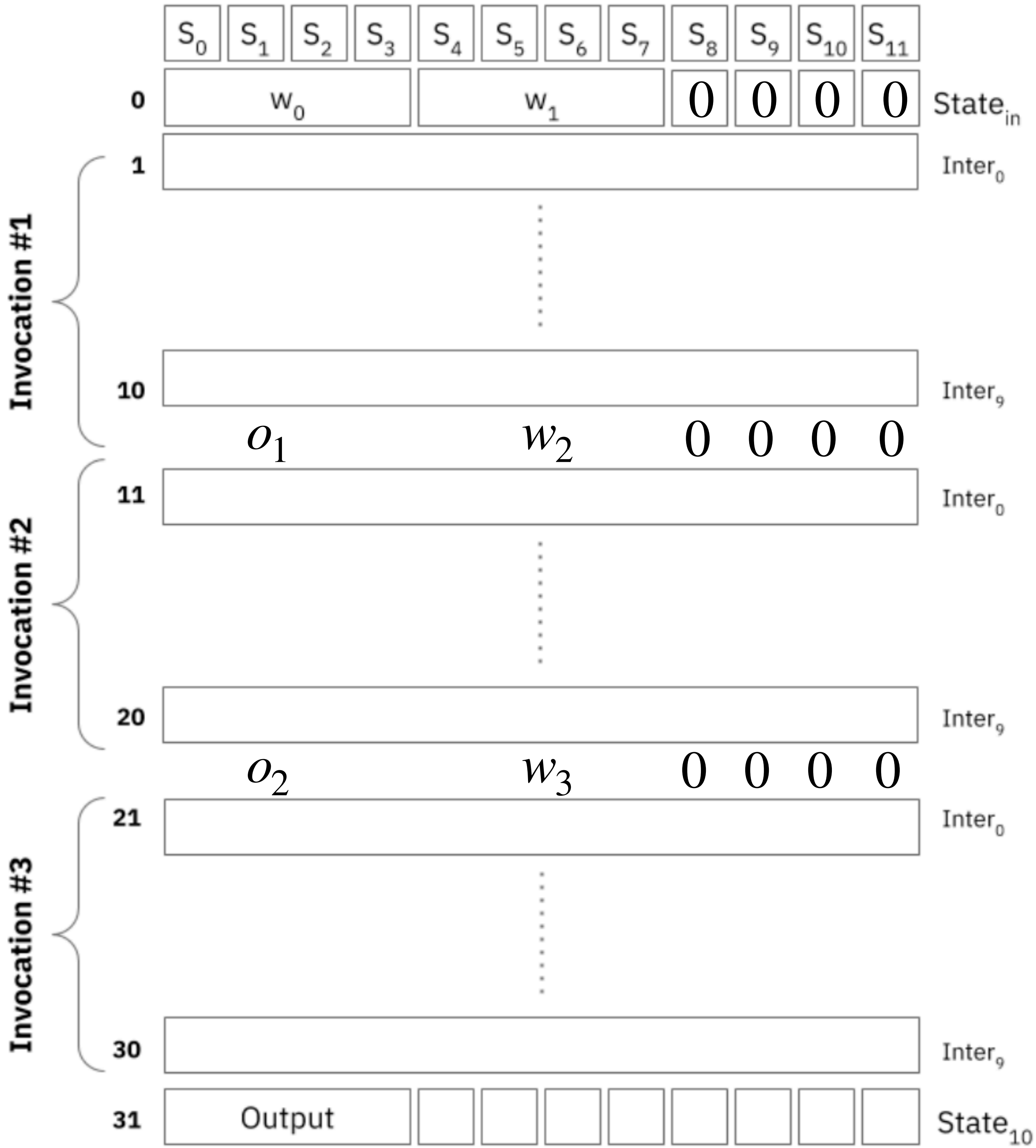
T: Trace evaluation domain

$g \in \mathbb{F}_p^{\times}$: multiplicative group of domain size T = 2^b , $b \in \mathbb{Z}$

Each column is interpreted as a polynomial evaluations $(g^i, f_j(g^i))$ of degree < T

There are S iNTTs of size T for interpolation.

Execution Trace in ethSTARK rescue



The State Inter is the state after end of Step 1 and beginning of Step 2

Hashes are executed and outputs recorded in batches of 3.

o_1 : digest at end of invocation 1 of rescue

o_1 : digest at end of invocation 2 of rescue

Output: digest at end of invocation 3 of rescue

The protocol is built for constraints on the trace to be checked by verifier.

ethSTARK complexity

Small field, field arithmetic and hash execution relatively fast,
Run time is not an issue

The main **computational bottlenecks are from Arithmetization**

Interpolation of AIR rep into polynomials, evaluation of constraint polynomial (FFT) : 75% of prover time!

Merkle Commitment (FRI) with Rescue : 20% of prover time!

What next?

Hashes in ZK

Hash function	Application	Use case
Poseidon [10]	Filecoin [11]	Vanilla Hash computation Octinary/Binary Merkle Tree computation Circuit: R1CS arithmetization
	Mina (Kimchi)* [12]	Turboplonk arithmetization, Fiat-Shamir
	Scroll tech [13, 14]	Plonkish Arithmetization (Halo2) Fiat-Shamir (Aggregator circuit)
Poseidon377	Penumbra* [15]	-
Poseidon-Goldilocks	Plonky2 [16]	Turboplonk arithmetization Merkle commitment, FRI-IOP [17]
	Polygon-zkEVM [18]	AIR arithmetization, FRI-IOP
Rescue [4], Rescue-Prime [19]	ethSTARK [4]	AIR arithmetization Merkle commitment, FRI-IOP
	Polygon MidenVM [20]	Vanilla hash computation Merkle tree, AIR arithmetization
Sinsemilla [21]	ZCash* [22]	Merkle Commitments, Lookup tables
Pedersen hash		Commitments, Merkle tree
Anemoi [23]	-	Optimized for Merkle trees (Anemoi-Jive)
Reinforced concrete [24]	-	Specialized for Look up tables
MIMC [25] Grendel [26] Griffin [27]	-	-

Table 1. Finite field friendly Hash functions in ZK space. The * refers to usage in sub-systems not in production at the time of writing of this paper.

Main Challenges

Many new hash functions which are friendly for ZK have been proposed: Such as Grendel, Griffin, they all seem to do some tradeoffs in this diagram by interpolating between HADES and MARVELlous

Low computational complexity

High computational complexity

It is all relative!

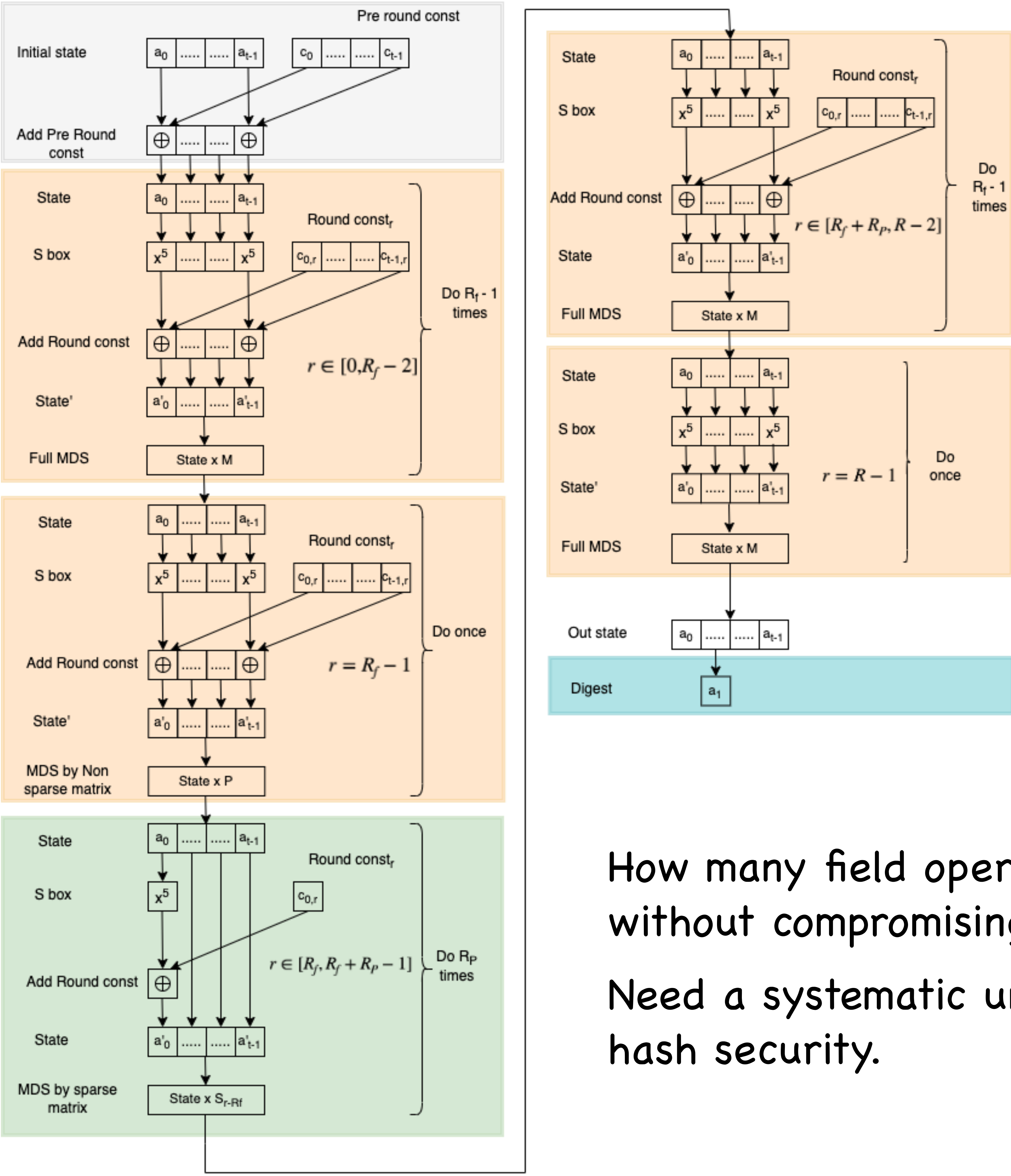
Less secure

More Secure

Reinforced concrete is optimized for lookups, but it is not clear how useful it is at this point.

The ZK hashes are relatively new and need to be battle tested for Security

From the computational complexity point of view, sometimes one can drop certain operations without compromising security



Miki's optimization: the digest size is one: then only need to evaluate that corresponding matrix element in the corresponding MDS

How many field operations can one reduce without compromising security?

Need a systematic understanding of ZK hash security.