SoK: Hash functions in Zero Knowledge Proofs

Karthik Inbasekar
karthik@ingonyama.com

Abstract: In this article we present a systematization of knowledge (SoK) on the usage of hash functions in Zero Knowledge Proofs (ZKP's). Since ZKP's operate on finite fields, traditional time-tested hash functions like SHA256 are unsuitable due to overhead in proving and verification of large number of bitwise operations. ZK-friendly hash functions are optimized for modular arithmetic in a finite field which reduces the circuit complexity significantly. This has led to widespread deployment of these hash functions in production-level systems. That said, the ZK-friendly hash functions are relatively newer than the traditional hash functions, and are yet to undergo the test of time. Our main goal is to survey such ZK-friendly hash functions, and highlight their general use cases in ZKP's.

Contents

1 Introduction **1**

2 Hashes in ZKP: A gentle introduction **4**
2.1 Universal properties 4
2.2 Permutation function 6

3 Use cases **8**
3.1 Merkle trees and commitments 8
3.2 Fiat-Shamir 9
3.3 Poseidon: Filecoin 10
3.4 Rescue: ethSTARK 12

4 Overview of ZK-friendly hashes in other applications **15**

5 Closing Words and a Disclaimer **16**

1 Introduction

A Zero Knowledge Proof (ZKP) is a protocol between a Prover and a verifier, which allows a prover to convince a verifier that a certain statement is true without revealing anything more than the truth of the statement itself. The statement takes the form of a NP relation $\mathcal{R}(x, w) : y = F(x, w)$ where x is a public input and w is a private input. The ZKP ensures that a verifier will be convinced by the proof of a honest prover who knows w, and learns nothing about w itself. While simultaneously ensuring that, a malicious prover who does not know w will be unable to convince the verifier with very high probability.

The NP relation is typically expressed as an arithmetic circuit, and there are several arithmetization programs available such as Rank One Constraint System (R1CS) [1, 2], Plonk Arithmetization [3] and AIR (Algebraic Intermediate Representation) arithmetization [4]. Arithmetic gates form the atomic structure of the circuits. Depending on the arithmetization, the spectrum of gates design in a circuit goes from simple addition or multiplication gates to a custom gate capable of performing compound operations. The input and output values of the gates are referred to as wire values. In ZKP, these wire values are elements in a finite field $\mathbb{F}$ (usually a large prime) and thus the fundamental computational primitive is modular arithmetic over a finite field.

Hash functions are versatile cryptographic primitives that are crucial in the design and functionality of many ZKP's. Many well known hash functions such as SHA2, SHA3, BLAKE are designed such that hashes can be computed blazingly fast on traditional CPUs. This is one of the reasons why these hash functions contain a lot of bitwise operations. While these are collision resistant and known to be secure, they are unsuitable for ZKP. This is because the prover will have to compute a lot of bitwise operations in a circuit[1] such as (1.1), and the verifier will have to check every bit with the solution, since the statement is only satisfied if all the computed bits are agreed upon (See for instance the SHA256 example in [5]). Thus, representing bit-wise operations of large bit-size numbers leads to large circuit sizes, which in turn leads to large polynomial degrees in the problem arithmetization. For instance the Discrete Fourier Transform complexity increases with the polynomial degree and directly affects the execution time of the prover. We caution the reader that the ZK space is moving fast, and this picture might very well change with the new developments in ZK that are based on lookup tables [6–8]. So far there are no production ready systems that efficiently employ lookup tables. For the purpose of this article we consider ZK-friendly hash functions that are natively defined over finite fields, have preimage resistance (one way property) and collision resistance. Below we summarize some of the use cases of hash functions in ZKP's.

- Cryptographic hash functions naturally appear in ZKP as the NP relation. The circuit could be made of a sequence of hash functions

$$H(\dots H(H(H(w_0, w_1), w_2), w_3) \dots, w_{r-1}) = y \tag{1.1}$$

  where the w_i's are private inputs of the prover while H and y are known and agreed upon by the prover and the verifier. The prover will claim knowledge of the preimage w_i, execute the circuit, and provides the verifier a cryptographic proof for computational integrity of the execution. A well known example is to provide proof of membership of a leaf in a Merkle tree. Generally, the hash function is represented as a circuit over a (large) prime field, and thus proving the knowledge of the preimage is computationally intensive. The main challenge is to find efficient representations of hash function in the circuit, to minimize arithmetic complexity such that it results in low degree polynomials and minimizes the number of constraints. (see for example [9]). These considerations lead to the search for "arithmetization oriented" hash functions where the efficiency efforts are primarily focused on optimization of algebraic

[1]Intuitively, the reason why bitwise operations are inefficient with arithmetic circuits is because a single AND or XOR operation would need one multiplication or addition gate. On the other hand, the same multiplication or addition gate is capable of multiplying or adding two very large numbers.

complexity as described in R1CS/AIR/Plonk constraints for ZKPs, minimizing the prover's computation. Thus unlike traditional hashes, the ZK friendly hashes are less focused on minimizing execution time of the hash operation itself.

- In addition, Hash functions are used as a building block of the cryptographic commitments based on Merkle trees, where the leaves of the Merkle tree are elements of some vector that one wants to commit to. The main computation in this case is a sequence of hashes, built in the form of a tree, with the root of the tree being the commitment to original data (see §3.1).
- Furthermore, Hash functions are also used as in the conventional way, e.g. while signing transactions or for authenticated encryption of fixed length messages or to generate pseudo-random field elements based on a seed. These predefined fixed-length sequence of inputs allow hash functions to instantiate a random oracle for Fiat-Shamir transform, removing interactivity between the prover and verifier (see §3.2).

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
Poseidon377	Penumbra* [13]	-
Poseidon-Goldilocks	Plonky2 [14]	Turboplonk arithmetization Merkle commitment, FRI-IOP [15]
	Polygon-zkEVM [16]	AIR arithmetization, FRI-IOP
Rescue [4], Rescue-Prime [17]	ethSTARK [4]	AIR arithmetization Merkle commitment, FRI-IOP
	Polygon MidenVM [18]	Vanilla hash computation Merkle tree, AIR arithmetization
Sinsemilla [19]	ZCash* [20]	Merkle Commitments, Lookup tables
Pedersen hash		Commitments, Merkle tree
Anemoi [21]	-	Optimized for Merkle trees (Anemoi-Jive)
Reinforced concrete [22]	-	Specialized for Look up tables
MIMC [23] Grendel [24] Griffin [25]	-	-

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.

Some of the finite field friendly hash functions that are used in the ZK space are listed in table 1. Our main goal in this article is to provide a survey of the ZK-friendly hash functions, and give some intuition about their use cases in ZKPs. In particular, we will defer the security and efficiency analysis of the hashes involved to a future article, since it deserves a more thorough technical treatment. The article is organized as follows. In §2 we start off with some universal properties of hash functions specific to ZK §2.1. In §2.2 we review the sponge construction and permutation which we believe are universal building blocks for ZK friendly hashes. In §3 we discuss some common use cases, including Merkle trees §3.1, and Fiat-Shamir heuristic for non-interactive protocols §3.2. Finally in §3.3, §3.4 we discuss two robust sponge constructions based on the HADES [26] and MARVELlous [27] designs, namely Poseidon hash [10] in filecoin [28] and Rescue hash [17] in ethSTARK [4]. Finally, in §4 we conclude with a high level overview of hash functions used in some of the Polygon ecosystems: Plonky2 [14], zkEVM [16] and Miden VM [18].

2 Hashes in ZKP: A gentle introduction

2.1 Universal properties

Generally speaking, hash functions in ZKPs operate on finite field elements. We observe the following characteristics in ZK-friendly hash functions:

- Natively defined to operate on elements in a finite field $\mathbb{F}$.
- Equipped with only addition, multiplication operations and potentially look-ups.
- Low arithmetic complexity (constraints, polynomial degrees)
- If the application requires b bit security, then the hash function must also be at least b bits secure. Furthermore, the usual properties of hash functions continue to hold.
  - **One way property**: Also called pre-image resistance, roughly defined such that given the output of a Hash $h = H(x)$ it should be computationally infeasible to invert the function and determine the pre-image x.
  - **Collision resistance**: It should be computationally infeasible to find two different x's that generate the same h.

All the hash functions listed in table 1 are based on the sponge construction [29]. Although the subject of hashes is by itself very rich and diverse, the specific requirements of ZKP's that we discussed in the introduction seems most suited for Substitution Permutation Networks (SPN's) based on the sponge construction. The sponge is an iterative construction that takes in arbitrary input/output sizes but applies a fixed length permutation f. The functional form of f can vary across hashes. However, typically f consists of some linear layers that include element-wise addition and matrix multiplication, and non-linear layers that consists of power maps. Without going into too much detail, the power maps are designed to increase the degree of the elements, while the linear layer spreads them uniformly across $\mathbb{F}$.

The sponge permutation has two parameters: the rate r and capacity c, that set the security level to be at least:

$$S = \log_2(\sqrt{p}) \min(r, c) \tag{2.1}$$

for preimage resistance and collision resistance, where p is the prime modulus. The instance of a sponge is specified by the triplet (t, p, S). Consider a tuple of field elements $t = r + c \in \mathbb{F}_p^{r+c}$, where r is the rate and c the capacity, while p is the prime modulus. The sponge permutation takes a tuple of size t, applies a series of fixed length $|t|$ permutations and outputs a digest r elements in $\mathbb{F}^r$. The rate r affects the throughput, and the capacity should be chosen such that (2.1) is met for the application in question. In fig 1, we quote the general definition of a sponge. In general the input message could be much larger than the permutation width $|t|$, in fig 1 the input message $M = [m_1||m_2||m_3||\ldots]$ is padded such that $|m_i| = r$. Then after each permutation, a part of the message M i.e m_i is "absorbed" into the permutation function. The initial state is chosen to be $I = [0^r||0^c]$. The sponge has two phases, absorb and squeeze.

- In the absorb phase, the f function is applied on the input $t = [m_i \oplus r_i||c_i]$. The permute block f outputs t elements which undergo further permutation after addition with the next message: $[m_{i+1} \oplus r_{i+1}||c_{i+1}]$, and so on. Once all m_i are exhausted, the sponge design switches to squeeze phase.

- The output at the end of the absorb phase has already reached the uniform distribution. In the squeeze phase, the user can determine the number of output blocks, and for each block the first r elements are outputed as the digest, followed by another permutation f until the number of output blocks is reached.

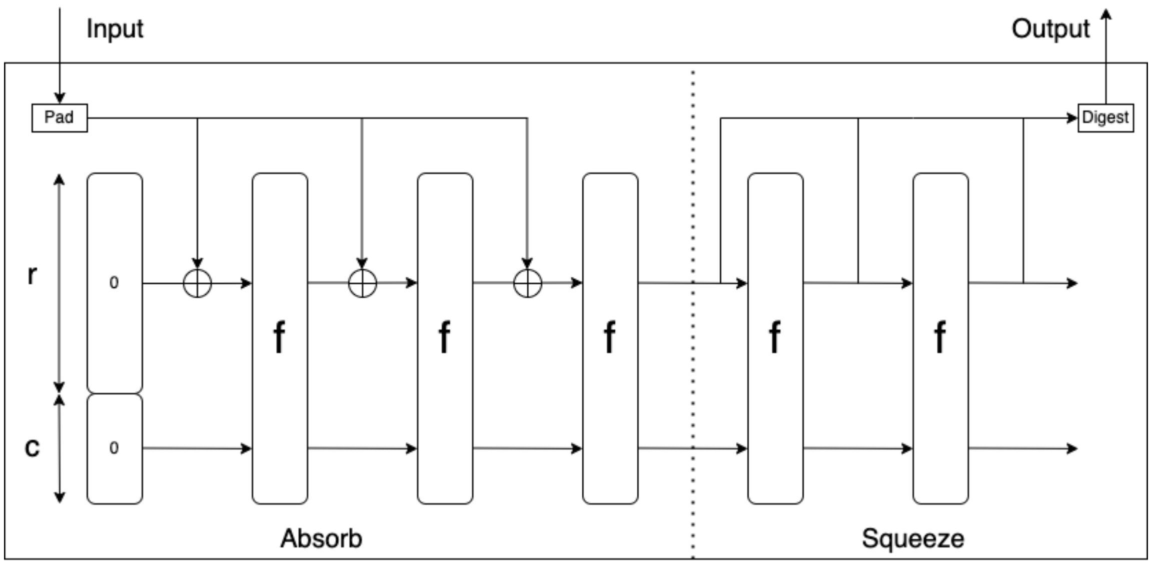

Figure 1. Sponge construction [30]

Note that the sponge method is quite like a symmetric cipher and can be used to encode arbitrary large messages as well. For ZKP's however, we do not worry about encoding large messages, but rather the input size M is known a priori, and the hash instance is either part of an arithmetic circuit for a NP statement such as (1.1), a standalone computation or a Merkle tree. In the following section we discuss the permutation function used in some of the hash functions keeping the above statement in mind.

2.2 Permutation function

Given an input state `state` $\in \mathbb{F}^t$, the permutation f is a bijective map $\mathbb{F}^t \to \mathbb{F}^t$. The permutation function (the hash instance), in general proceeds over several rounds R in a sequence of operations. The requirement of R is largely fixed by satisfying security inequalities and vary across implementations.

1. **Instantiation:** We instantiate the hash by a parameter triple (t, p, S) which sets the prime field, the security level, and the size of the internal state $|t|$.
   - **The prime field modulus** p**:** in bit size $n = log_2(p)$
   - **Security level** S**:**
   - **The width** t**:** is calculated using the formula

   $$t = \mathbf{len}(\text{input}) + \mathbf{len}(\text{output}) = \mathbf{len}(\text{input}) + \left\lceil \frac{2S}{\log_2(p)} \right\rceil \tag{2.2}$$

   As an e.g. for a hash with 128 bit security, and a 256 bit prime field, the digest size is 1.
2. **Preparatory:**. From the instantiation (t, p, S) all additional parameters required for the hash are computed. We will represent all intermediate states of the hash by the variable

   $$\mathsf{state}[i]\ ,\ \forall i = 0, 1, \ldots, t-1$$
3. **Computation:** The computations consists of sequential operations referred to as the round function. At the end of all the rounds R the hash function outputs the digest element(s).

In general the round function consists of the following building blocks.

1. **Non-linear layer (S-box)**: The S-box is chosen as the power map

   $$\pi_0 : x \to x^{\alpha} \tag{2.3}$$

   $$\pi_1 : x \to x^{-1} \tag{2.4}$$

   where $\alpha \geq 3$ is usually chosen as the smallest integer that guarantees invertibility and provides non-linearity. The S box exponent α is the smallest possible integer that satisfies

   $$\gcd(\alpha, p-1) = 1 \tag{2.5}$$

   where p is the prime modulus in a prime field $\mathbb{F}_p$. If no such α is found satisfying (2.5), then one can use π_1. Depending on the design of the hash sometimes either or both of (2.3), (2.4) are used. The main reason to include the power map, is to increase the degree such that the security requirements are met.

2. **Addition of Round Constants**: This is simply t field additions of the state with that round's constant RoundConstants_j.

$$\texttt{state} = \texttt{state} \vec{\bigoplus} \text{RoundConstants}_j \tag{2.6}$$

The round constants play the role of the public key, and usually new values are injected into each round. Given the triple (t, p, S) these keys are generated usually in a deterministic way and known a-priori the hash computation itself.

3. **Linear layer**: The linear layer is a matrix multiplication of the state by an MDS (Maximum Distance Separable) matrix whose goal is to spread the uniform randomness properties across the entire state.

$$\texttt{state} = \texttt{state} \times \mathcal{M} \tag{2.7}$$

The MDS matrix also depends only on the triple (t, p, S) and is precomputable prior to the hash invocation.

Below we try to give a flavor of the definitions above for two hash functions permutations designs, one is based on HADES [26] and the other is based on MARVELlous [27]. The main difference between the two designs is in the non-linear layer as explained below (also see fig 2)

- In the HADES design [26], the S-boxes are unevenly distributed across several rounds. There are partial rounds where the S-box is only applied to some of the elements in t, and full rounds where the S-box is applied to all elements in t. The structure of the rounds are such that the partial rounds are sandwiched in one batch in-between two batches of full rounds. The external rounds protect against statistical attacks and the internal rounds raise the degree of the permutations. An example of this design are Poseidon/Reinforced concrete [10, 22].

- In the MARVELlous design [27], the non-linear layer uses the power map π_1 (2.3) in even rounds and π_2 (2.4) in odd rounds. Since 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. This increases the computational complexity of the hash, but provides better security. An example of this design is the Rescue/Rescue-prime [4, 17].

We wish to stress that despite several new hash functions such as Anemoi [21], Grendel [24] and Griffin [25], the basic operations are some variations of the HADES or MARVELlous templates. Note that reinforced concrete [22] and Sinsemilla [19] are the first of the ZK-friendly hashes optimized for lookup tables. In section 3, we describe some of the general use cases of hashes in ZK, followed by a review of Poseidon and Rescue from the HADES and MARVELlous families respectively.

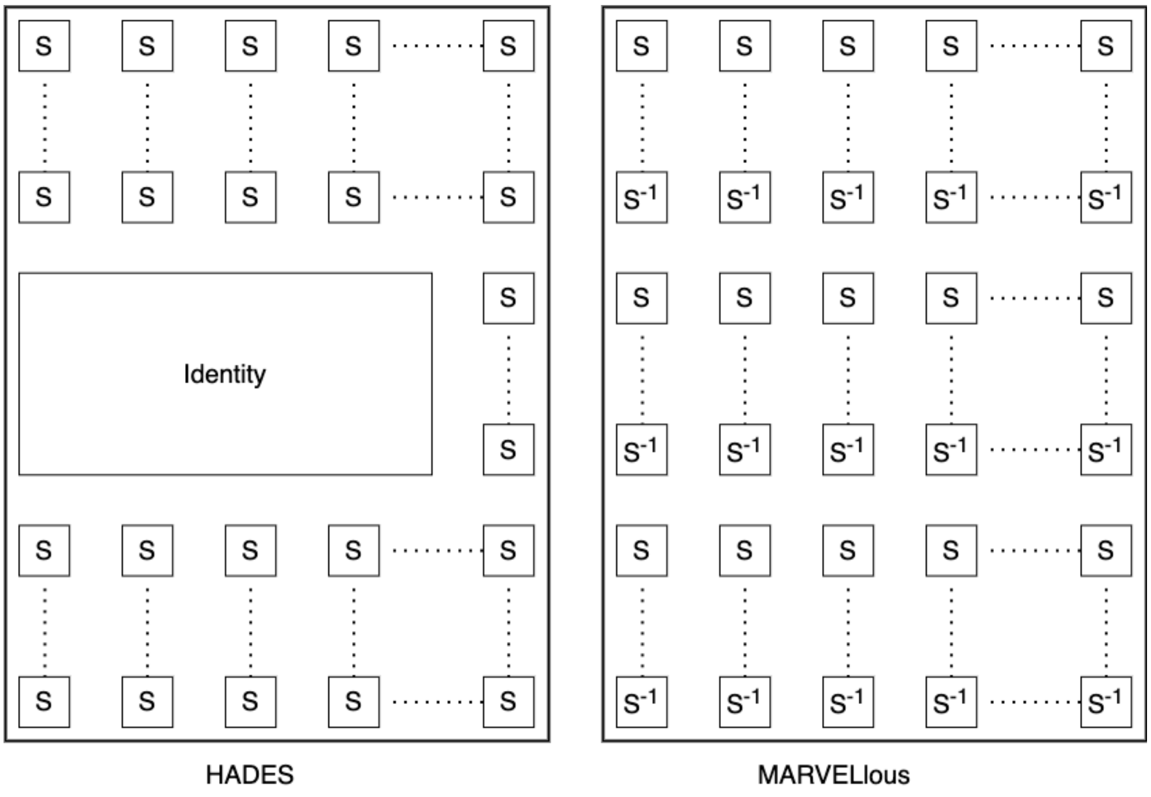

Figure 2. HADES vs MARVELlous design: The rows on each represent the states, S represents the map π_1 (2.3) and S^{-1} represents π_2 (2.4). In HADES there are uneven distribution of S-boxes. The partial rounds are sandwiched between full rounds. In each partial round, the S-box is applied to one element in t, and identity for the remaining $t-1$ elements. For the MARVELlous design, the S-box π_1 and the inverse S-box π_2 are applied in every round.

3 Use cases

In this section we discuss some straightforward usages of hashes in ZK space, such as merkle commitments §3.1 and Fiat-Shamir §3.2 transformations and more specific applications of Poseidon §3.3 and Rescue §3.4 where arithmetization of the hash plays a more significant role.

3.1 Merkle trees and commitments

A Merkle tree is a data structure of nodes, in a layered tree form. See for instance fig 3 for a binary merkle tree. The nodes in the bottom most layer are called the leaf nodes H_i, which in general contain hashes of data. Each non leaf parent node is a hash of its child nodes. The one way property and collision resistance of the hash function guarantees that the sequence of such hashes into a unique root thus making it a binding commitment scheme for any vector of nodes. Each leaf element in the tree has a unique path that specifies its route and siblings on the way to the root. For e.g. in fig 3 the leaf T_2 has the path element

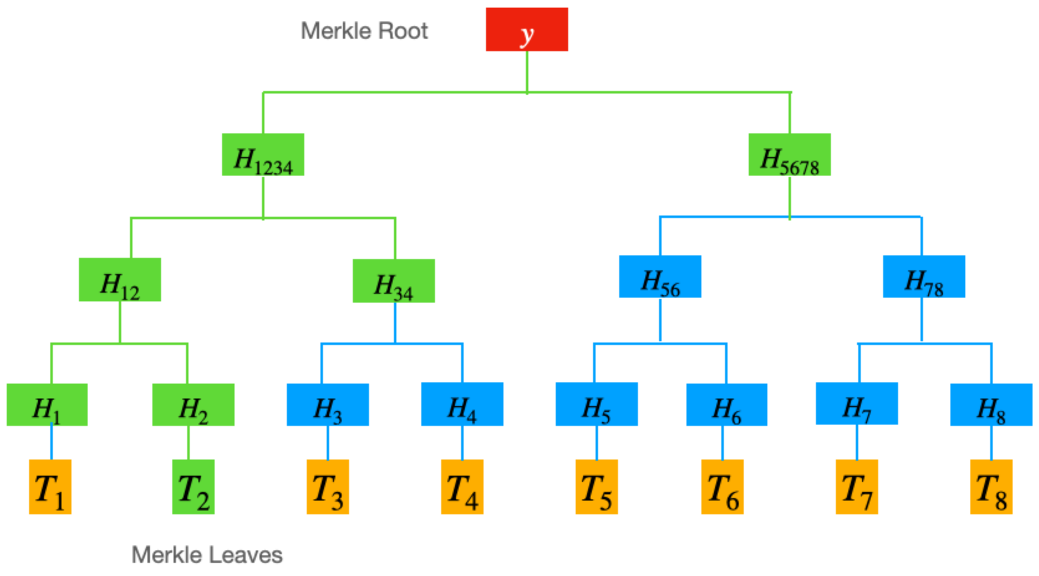

Figure 3. A binary Merkle tree: $H_{ij} = H(H_i, H_j)$, the blocks in green are the path elements corresponding to proof of membership of T_2 in the tree. The root y is a public commitment.

(L, L, R) from y and has the siblings $\Pi : (H_{1234}, H_{5678}, H_{12}, H_{34}, H_1, H_2)$. [2]. As one can see in fig 3, the tree has $n_{nodes} = 8$, the depth of the tree is $3 = \log_2(n_{nodes})$. In order to prove that T_2 is an element in the leaf of the tree with y a public commitment, the proof should consist of $3 * 2 = 6$ elements.

In a more general case, where instead of a binary tree, if we have a t-nary tree, it would have a depth $d = \log_t(n_{nodes})$ and the total proof size for membership in the leaf would amount to $d * t$ field elements.

3.2 Fiat-Shamir

The Fiat–Shamir heuristic allows one to take an interactive argument of knowledge and uses random oracle properties to convert it into a non-interactive argument of knowledge. As the reader would have guessed, the hash function here plays the role of the random oracle and creates a digital signature of the interaction. Consider the following sequence of interactions (based on [31])

1. The prover and verifier agree on a public data $x \in \mathbb{F}^{|x|}$
2. Prover adds witness, does a prescribed computation and sends proof elements $\Pi_a \in \mathbb{F}^{|t|-|x|}$
3. Verifier responds with challenge $\alpha \leftarrow \mathbb{F}$
4. Prover includes α in a prescribed computation and responds with proof element $\Pi_b \in \mathbb{F}^{|t|-|x|}$

[2]L: left, R: Right

5. Verifier responds with challenges $\beta \leftarrow \mathbb{F}$
6. Prover includes α, β in a prescribed computation and responds with final proof Π_c

The protocol is rendered non-interactive by replacing the interaction steps $3, 5$ with a random oracle hash $c \leftarrow H(a_0, a_1, \ldots a_{t-1})$ as follows.

1. The prover and verifier agree on a public data $x \in \mathbb{F}^x$
2. Prover adds witness, does a prescribed computation and computes proof $\Pi_a \in \mathbb{F}^{|t|-|x|}$
3. Prover computes $\alpha \in \mathbb{F}$, by $\alpha \leftarrow H(x, \Pi_a)$
4. Prover includes α in a prescribed computation and generates proof element $\Pi_b \in \mathbb{F}^{|t|-|x|}$
5. Prover computes $\beta \in \mathbb{F}$ by $\beta \leftarrow H(x, \Pi_a, \Pi_b)$[3]
6. Prover includes α, β in a prescribed computation and responds with final proof Π_c

If both the prover and the verifier agree on the random oracle generator, i.e the hash function and how it is computed a priori, then the protocol becomes non-interactive. Note that the form of the hash does not just include the values sent by the prover to the verifier in the interactive protocol, but also includes the public value. This is necessary to prevent the "frozen heart" vulnerability that allows a malicious prover to forge proofs [32]. Almost every ZKP implementation uses the Fiat-Shamir Heuristic, and although it is not necessarily a computational bottleneck, the hashes used in this part need to be secure.

3.3 Poseidon: Filecoin

In this section, we describe Poseidon [10] as it is used in the Filecoin project [28]. Following our discussion of the universal properties of hashes in ZKP §2.1 and more specifically the sponge construction in §2.2, we define a poseidon instantiation as specification of the triple (t, p, S) where p is the prime modulus of the scalar field. Filecoin uses the elliptic curve BLS12-381 [33]. The parameters of the scalar field in the group are

$$\begin{aligned} \text{bit size} &= \log_2(p) \sim 256 \text{ bits} \\ \text{Security} : S &= 128\text{bits} \\ \text{digest size} &= \left\lceil \frac{2S}{\log_2(p)} \right\rceil = 1 \end{aligned} \tag{3.1}$$

The computation consists of two types of rounds of sequential operations referred to as the round function.

- R_F Full rounds represented split into two $R_f = R_F/2$ rounds.
- Partial rounds R_p which operates between the two R_f rounds.

[3]It is really important to include Π_a in this follow up challenge, this is necessary for proof non-malleability via challenge generation. We thank Suyash Bagad for pointing this out.

At the end of all the rounds $R_f + R_P + R_f$ the Poseidon hash function outputs the digest $\texttt{state}[1]$. Any round function of Poseidon permutation consists of the following three components.

1. *Add Round Constants*, denoted by $ARC(\cdot)$. Every round j begins with t field additions of the state with that round's constant RoundConstants_j.
$$\texttt{state} = \texttt{state} \vec{\bigoplus} \text{RoundConstants}_j \tag{3.2}$$
2. The non-linear layer: *S-box function* is defined as
$$\begin{aligned} S : \mathbb{F}_p &\to \mathbb{F}_p \\ S(x) &= x^5 \end{aligned} \tag{3.3}$$
If the round corresponds to a full round, the S-box is applied on each element of the state, else only on the first element
$$\texttt{state} = \begin{cases} \texttt{state}[i]^5\big|_{i=0,1,\ldots,t-1} & \text{Full round} \\ \texttt{state}[0]^5 & \text{Partial round} \end{cases} \tag{3.4}$$
The total number of S boxes is given by $N_{S-box} = tR_F + R_p$ and is optimized while satisfying the security inequalities (see Poseidon paper [10]).
3. MDS layer denoted by $\mathcal{M}_{t \times t}(\cdot)$: This operation is the linear layer of Poseidon construction. It consists of the matrix multiplication of the $\texttt{state}$ by a $t \times t$ MDS matrix.
$$\texttt{state} = \texttt{state} \times \mathcal{M} \tag{3.5}$$

For more details, on the MDS matrix and round constant generation we refer to our github implementation in Python [34]. The Poseidon instantiations used in filecoin are summarized in table 2. Let us briefly discuss how the Poseidon function is used in filecoin. The relevant

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

process is known as PC2, and we refer to the documentation [28] for further details. What is relevant for this discussion is that, the data structure in $PC2$ of filecoin consists of 11×2^{30} array of 256 bit field elements. $\texttt{Poseidon}_{11}$ is used to hash all the columns into digests that create the leaves of a Merkle Tree. Following which $\texttt{Poseidon}_8$ is employed to create an Octinary Merkle tree of depth $\log_8(2^{30}) = 10$. In addition another instance of $\texttt{Poseidon}_8$ is used on the last layer of the data structure to create another Octinary Merkle tree. Finally the digests of the two Octinary Merkle trees are hashed using $\texttt{Poseidon}_3$ and the digests are committed publicly. The total number of Poseidon hashes is roughly $2^{30} + (2^{30} - 1)/7 * 2 + 1$ taking into account the column hashes and the Merkle trees. It takes about $6 - 7$ minutes using a $RTX3090$ GPU to just run the hash execution part of the protocol. Thus this is a rather unique case, where the execution of the hashes themselves take time. This is perhaps already evident in table 2 where we find that there are substantial modular addition and multiplication operations on 256 bit field elements.

In order to provide a proof of integrity of the 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. The R1CS arithmetization of the Poseidon hash function creates gates with "free additions" built in due to large fan in support of the R1CS system. In table 3 we list the sizes of the constraints in R1CS arithmetization for the knowledge of preimage problem in the case of $\texttt{Poseidon}_3$ and $\texttt{Poseidon}_{12}$, and knowledge of membership problem in the case of the Merkle tree instances $\texttt{Poseidon}_8$. The circuit sizes for the entire

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} [25]

Filecoin project and the arithmetization leads to polynomials of sizes up to 2^{26}, and it takes roughly 7 minutes to generate a Groth16 Proof using a $RTX3090$ GPU. Thus in this use case, both the evaluation of the hashes and the generation of the ZKP are compute bottlenecks.

3.4 Rescue: ethSTARK

In this section, we describe Rescue/Rescue-prime [17] as it is used in the ethSTARK project [4]. The Rescue/Rescue-prime hash is based on the MARVELlous design [27] discussed earlier. Unlike the filecoin project, ethSTARK is based on the Zk-STARKS and operates on a small 61 bit prime field

$$\begin{aligned} \text{bit size} &= \log_2(p = 2^{61} + 20 * 2^{32} + 1) \sim 61 \text{ bits} \\ \text{Security} : S &= 128\text{bits} \\ \text{digest size} &= \left\lceil \frac{2S}{\log_2(p)} \right\rceil = 4 \end{aligned} \tag{3.6}$$

The initial state consists of field elements $[\mathbb{F}^4 || \mathbb{F}^4 || 0^4]$ and the round consists of two batches, with each batch differing in the non-linear layer (S-box). Besides the S box is now applied to all elements in all rounds.

1. Batch1: The first non linear layer begins with a cube root permutation
$$\pi_1 : x^{1/3} \equiv x^{\frac{2p-1}{3}} \tag{3.7}$$
in this case the equivalence in the RHS above is due to the fact that $3 \nmid p-1$. This is followed by a MDS matrix multiplication
$$\texttt{state}_{12} = \texttt{state}_{12} \times \mathcal{M}_{12\times 12} \tag{3.8}$$
and Round constant addition
$$\texttt{state} = \texttt{state} \vec{\oplus} K_r \tag{3.9}$$
to generate an Intermediate state `Inter` which is used for measuring AIR constraints later on.

2. Batch2: consists of a non linear layer that cubes the state
$$\pi_2 : x^3 \tag{3.10}$$
followed by a MDS matrix multiplication
$$\texttt{state}_{12} = \texttt{state}_{12} \times \mathcal{M}_{12\times 12} \tag{3.11}$$
and Round constant addition
$$\texttt{state} = \texttt{state} \vec{\oplus} K_r \tag{3.12}$$
to end the round.

The sequence of operations is summarized in fig 4. For 128 bit security the Rescue security inequalities require about 10 rounds till the desired goal of uniformity is achieved [4].

The circuit for the ethSTARK is essentially that the prover knows a sequence of witness $w = \{w_0, w_1, \ldots w_n\}$ where $w_i \in \mathbb{F}^4$ such that (1.1) is satisfied. The hash chain length is $n = |w| - 1 = 3 * 2^i$ for $i \in [10, 18]$. Given the length of the hash chain, the prover runs the chain by providing the witnesses and recording the intermediate states `Inter` in an Algebraic Execution Trace (AET) as shown in fig 5. In the fig $g \in \mathbb{F}_p^\times$ (the multiplicative group of size T), and each column is interpreted as polynomial evaluations $(g^r, f_j(g^r))$ of degree $< |T|$. As we mentioned earlier each invocation of the hash takes about 10 rounds and generates 10 of the `Inter` AET rows, and one must add the initial and final state as well. The AET together with the constraints on the various rows and columns form the AIR arithmetization of the computation.

For technical reason the ethSTARK runs invocations in multiples of 3, thus the AET is of dimension $2^{i+5} \times 12$. The execution of these hashes and run times are relatively fast and are not a major compute bottleneck, presumably because of the small bit prime fields. The main compute bottleneck in the STARK case comes because the AET needs to be converted into a polynomial using iNTT (Inverse Number Theoretic Transform) [4], after that they are

[4]Inverse Discrete Fourier Transform in a finite field

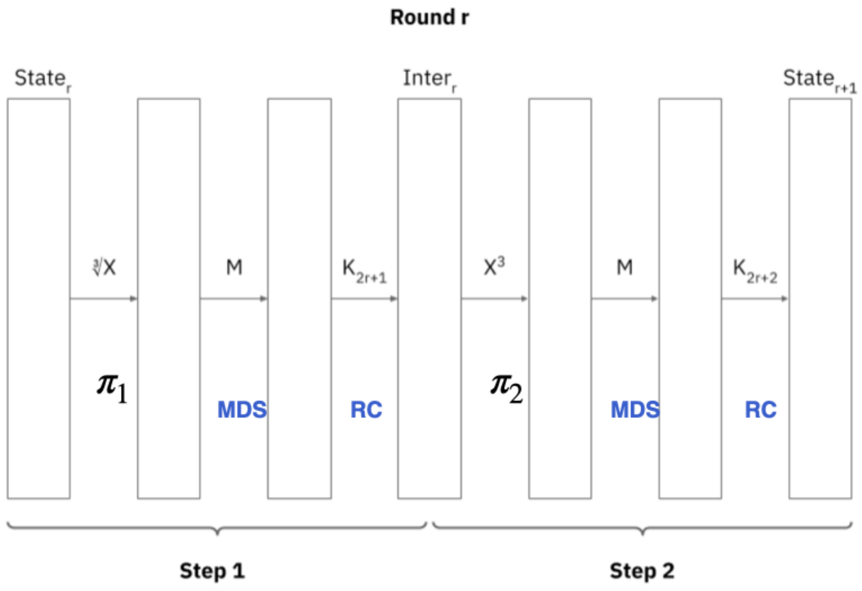

Figure 4. Round function in Rescue

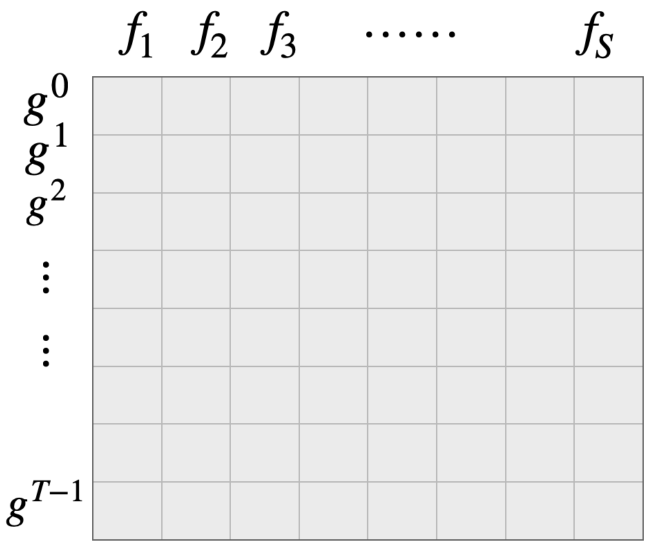

Figure 5. Algebraic execution trace : AET

evaluated in an extended domain which includes a blow up factor $\beta \in [4, 8, 16]$ resulting in polynomial degrees up to $\beta * 2^{i+5}$ which can be very large. The NTT/iNTT takes about 80% of the computational time in the ethSTARK system. Furthermore, the proof system uses FRI-IOP protocol which requires the prover to Merkle commit polynomials of a given degree N in $\log_2 N$ steps, where in each step the degree is decimated by 1/2. While we discussed the Merkle commitment §3.1 earlier and observed that it is inefficient, in ethSTARK it consumes up to about 20% of the prover time and is not negligible. This is example of a ZK friendly hash in a STARK system, where both NTT/iNTT and Merkle commitments/FRI are computational bottlenecks.

4 Overview of ZK-friendly hashes in other applications

In this section, we review some of the other use cases not covered in detail in this article. In table 1 we mentioned the ZK applications from Polygon: zkEVM [16], Plonky2 [14] and MidenVM [18]. All these systems use the same prime field $\mathbb{F}_p$, where $p = 2^{64} - 2^{32} + 1$ corresponds to the Goldilocks prime[5]. In this section, we review some of the use cases of Zk-friendly hashes in these ecosystems.

In the case of Plonky2 [14], the Poseidon instantiation is called Poseidon-Goldilocks, the parameters are $(t = 12, p, S = 128)$ and the hash operates with the round parameters $(R_F, R_P) = (8, 22)$. Notice that the number of rounds is far less than the filecoin case discussed in table 2. However, in the non-linear layer, the S-box is $\pi_0 : x^7$ which is larger than the $\pi_0 : x^5$ used in the Filecoin instantiation. In our trial runs, 40% of the prover time is spent on the FRI-IOP in generating Merkle trees, with the remaining time spent in NTT/iNTT on permutation constraints in the Plonk [3] prover system. In using the Plonk prover, one has a freedom to arithmetize the hash function using many basic gates, or using custom gates for more complex functions. Using more basic gates, increases the number of rows in the execution trace and adds to prover complexity. While adding more complex custom gates increases the number of columns in the execution trace, and makes the constraints more complex, thereby increasing verifier time. Plonky2 uses a single custom gate to evaluate an entire instance of Poseidon, this results in execution traces with large number of columns, but fast prover times of about $300-350$ milliseconds. Since the proving technique used in Plonky2 is FRI, it results in long proof sizes, and Plonky2 uses Proof recursion to compress the proof sizes to about 43 kilobytes. The trade off between row vs column ratio in the Plonk execution trace, is an optimization unavailable in the R1CS or AIR arithmetization programs and is a topic of active research.

In Polygon zkEVM [16], the instantiation of the Poseidon hash function is identical to that used in Plonky2 [14] described in the paragraph above. The zkprover is a component of the EVM that has a state machine (VM) dedicated to Poseidon. The zkprover's state machines execute programs, and prove that the programs were correctly executed. The Poseidon State Machine (SM) records internal permutations (within the rounds of Poseidon) as state transitions and stores them in the form of rows of a lookup table. It further builds a polynomial constraint system for the state transitions, based on the Poseidon internal permutations. Furthermore, it expresses the state transitions as polynomials and commits to these polynomials using a Polynomial commitment scheme. Since the execution trace is now used as a lookup table, the zkprover uses Plookup [6] to verify if the committed polynomials satisfy the necessary constraints and produce correct trace entries in the lookup

[5]A Goldilocks prime has the structure $p = \phi^2 - \phi + 1$, where $\phi = 2^n$, In the context of the current discussion $\Phi = 2^{32}$. The main advantage of using the Goldilocks prime appears to be fast Karatsuba multiplications [35]

table. The zkprover has a STARK recursion component that operates STARK proofs using FRI-IOP. In the first step, it generates a STARK proof per execution of Poseidon VM. Following which, it batches a fixed number of zk-STARK proofs each corresponding to different executions of the Poseidon VM and produces a STARK proof per batch. Then it bundles a fixed number of batches of proofs and proceeds recursively to produce a single STARK proof, that proves all the other STARK proves that it represents as well as the VM executions represented by the proofs. Finally, the zkEVM compresses the STARK proof using a SNARK by employing the rapidsnark library [36]. This is yet another fascinating ecosystem where ZK-friendly hashes play a significant role.

Next we move on to Miden VM [18] which produces a STARK proof for any program executed by the system. The execution proof can be verified without knowing what the program is. Miden VM operates on the same Goldilocks prime field of plonky2 and zkEVM discussed above, and uses AIR arithmetization. The VM has a dedicated hash component to accelerate the computations of the ZK-friendly hash Rescue-prime [17]. More specifically, the "hash chiplet" in Miden VM is a processor that can execute instructions set (similar to hardware instruction sets for processors) to perform atomic operations in a hash, such as

- A single permutation of Rescue-prime (for e.g. a single round function in fig 4), and outputs the state at the end of a round.
- Execution of a simple $2 \to 1$ hash, i.e. $H(w_0, w_1) \to w_3$ where $w_i \in \mathbb{F}^4$ and H is the Rescue-prime hash function.
- Perform a linear hash of n elements in the sponge mode. The hash chiplet sets the rate of the permutation function to 8 and capacity to 4, and absorbs the n elements as described in fig 1. In the squeeze phase it outputs 4 field elements.
- Verification of path in a merkle tree (see fig 3) and to update root in the tree.

In all cases, the VM generates execution trace for AIR depending on the choice of internal states to define the constraint. The STARK proof is generated using the FRI-IOP protocol.

5 Closing Words and a Disclaimer

Our discussion leads us to a general picture that there is much to be explored in lowering the computational overhead of ZK-friendly hash functions. While security is paramount in the definition of hashes in general, low computation complexity is vital for scaling of any ZKP application. Since computational complexity scales with security in general, it is vital that the design of ZK-friendly hash functions add yet another component to their toolbox: Hardware friendliness.

Disclaimer: The ZK space is a fast moving target. While we try to keep this SoK up-to-date, note that the validity of some of the data presented here may change faster.

Acknowledgments

We would like to thank Suyash Bagad and Omer Shlomovits for a thorough review of the draft. We would also like to thank Yuval Domb, Oren Margalit, Ekaterina Semenova, and Guy Weissenberg for comments and suggestions. We thank Robin Salen, and Izzy Meckler for references.

References

[1] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps." Cryptology ePrint Archive, Paper 2012/215, 2012. https://eprint.iacr.org/2012/215.

[2] V. Buterin, "Quadratic arithmetic programs from zero to hero." https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649.

[3] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge." Cryptology ePrint Archive, Report 2019/953, 2019. https://ia.cr/2019/953.

[4] StarkWare, "ethstark documentation." Cryptology ePrint Archive, Paper 2021/582, 2021. https://eprint.iacr.org/2021/582.

[5] DECENTRIQ, "Proving sha256 preimage knowledge using zokrates." https://blog.decentriq.com/proving-hash-pre-image-zksnarks-zokrates/.

[6] A. Gabizon and Z. J. Williamson, "plookup, a simplified polynomial protocol for lookup tables." Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[7] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, "Caulk: Lookup arguments in sublinear time." Cryptology ePrint Archive, Paper 2022/621, 2022. https://eprint.iacr.org/2022/621.

[8] J. Posen and A. A. Kattis, "Caulk+: Table-independent lookup arguments." Cryptology ePrint Archive, Paper 2022/957, 2022. https://eprint.iacr.org/2022/957.

[9] ZPrize, "Accelerating poseidon hash function." https://assets.website-files.com/625a083eef681031e135cc99/6305a4714b50ef347bcd5ee3_poseidon.pdf.

[10] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems." Cryptology ePrint Archive, Paper 2019/458, 2019. https://eprint.iacr.org/2019/458.

[11] Filecoin, "Neptune." https://github.com/filecoin-project/neptune.

[12] Mina, "Kimchi." https://o1-labs.github.io/proof-systems/specs/kimchi.html#poseidon-hash-function.

[13] Penumbra, "Poseidon377." https://protocol.penumbra.zone/main/crypto/poseidon/paramgen.html.

[14] Polygon-Zero, "Plonky2." https://github.com/mir-protocol/plonky2.

[15] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast reed-solomon interactive oracle proofs of proximity." https://eccc.weizmann.ac.il/report/2017/134/.

[16] Polygon-Hermez, “Polygon zkevm.” https://github.com/0xpolygonhermez.

[17] A. Szepieniec, T. Ashur, and S. Dhooghe, “Rescue-prime: a standard specification (sok).” Cryptology ePrint Archive, Paper 2020/1143, 2020. https://eprint.iacr.org/2020/1143.

[18] Polygon, “Miden vm.” https://maticnetwork.github.io/miden/design/chiplets/hasher.html.

[19] ZCash, “Sinsemilla hash.” https://zcash.github.io/halo2/design/gadgets/sinsemilla.html.

[20] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, “Zcash protocol specification.” https://zips.z.cash/protocol/protocol.pdf.

[21] C. Bouvier, P. Briaud, P. Chaidos, L. Perrin, and V. Velichkov, “Anemoi: Exploiting the link between arithmetization-orientation and ccz-equivalence.” Cryptology ePrint Archive, Paper 2022/840, 2022. https://eprint.iacr.org/2022/840.

[22] L. Grassi, D. Khovratovich, R. Lüftenegger, C. Rechberger, M. Schofnegger, and R. Walch, “Reinforced concrete: A fast hash function for verifiable computation.” Cryptology ePrint Archive, Paper 2021/1038, 2021. https://eprint.iacr.org/2021/1038.

[23] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, “Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity.” Cryptology ePrint Archive, Paper 2016/492, 2016. https://eprint.iacr.org/2016/492.

[24] A. Szepieniec, “On the use of the legendre symbol in symmetric cipher design.” Cryptology ePrint Archive, Paper 2021/984, 2021. https://eprint.iacr.org/2021/984.

[25] L. Grassi, Y. Hao, C. Rechberger, M. Schofnegger, R. Walch, and Q. Wang, “A new feistel approach meets fluid-spn: Griffin for zero-knowledge applications.” Cryptology ePrint Archive, Paper 2022/403, 2022. https://eprint.iacr.org/2022/403.

[26] L. Grassi, R. Lüftenegger, C. Rechberger, D. Rotaru, and M. Schofnegger, “On a generalization of substitution-permutation networks: The hades design strategy.” Cryptology ePrint Archive, Paper 2019/1107, 2019. https://eprint.iacr.org/2019/1107.

[27] T. Ashur and S. Dhooghe, “Marvellous: a stark-friendly family of cryptographic primitives.” Cryptology ePrint Archive, Paper 2018/1098, 2018. https://eprint.iacr.org/2018/1098.

[28] Filecoin, “The filecoin project.” https://github.com/filecoin-project.

[29] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, in *Advances in Cryptology – EUROCRYPT 2008* (N. Smart, ed.), (Berlin, Heidelberg), pp. 181–197, Springer Berlin Heidelberg, 2008.

[30] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer, “Keccak hash.” https://keccak.team/sponge_duplex.html.

[31] D. Khovratovich, J. Aumasson, and P. Quine, “Safe (sponge api for field elements) – a toolbox for zk hash applications.” https://hackmd.io/bHgsH6mMStCVibM_wYvb2w?view.

[32] J. Miller, “Serving up zkps.” https://blog.trailofbits.com/2021/02/19/serving-up-zero-knowledge-proofs/.

[33] Arkworks, “Arkworks bls12-381.” https://docs.rs/ark-bls12-381/latest/ark_bls12_381/.

[34] E. Semenova, “Poseidon - python.” https://github.com/ingonyama-zk/poseidon-hash.

[35] J. Cook, "Goldilocks prime." https://www.johndcook.com/blog/2019/05/12/ed448-goldilocks/.

[36] Polygon, "Rapid snark." https://github.com/iden3/rapidsnark.