Evaluating ZKP Friendly Hash Functions for ZKFlow

June 28, 2022

Abstract

1 Introduction

In ZKFlow, we currently use Blake2s to compute the component leaf hashes since they have variable length preimages. We compute the rest of the tree using Pedersen hash on fixed length input with the same personalization value. Figure 1 shows the structure of the Merkle tree in the current design. Blake2s is chosen due to its efficiency compared to other standard hash algorithms and its resistance against length extension attacks. Despite its efficiency compared to other standard hash algorithms, the computation cost of Blake2s is still significantly higher in ZKP circuits (around 21K constraints in R1CS). In contrast to Blake2s, Pedersen hash enables efficient computations in ZKP circuits with around 1.3K constraints on a single hash. However, its weaker security guarantees and slower performance on plaintext limits the adoption of Pedersen hash as an alternative to Blake2s. The performance out of the circuit is crucial for us since it degrades the performance Corda's transaction Merkle tree calculation, which in turn significantly degrades the performance of Corda's backchain validation.

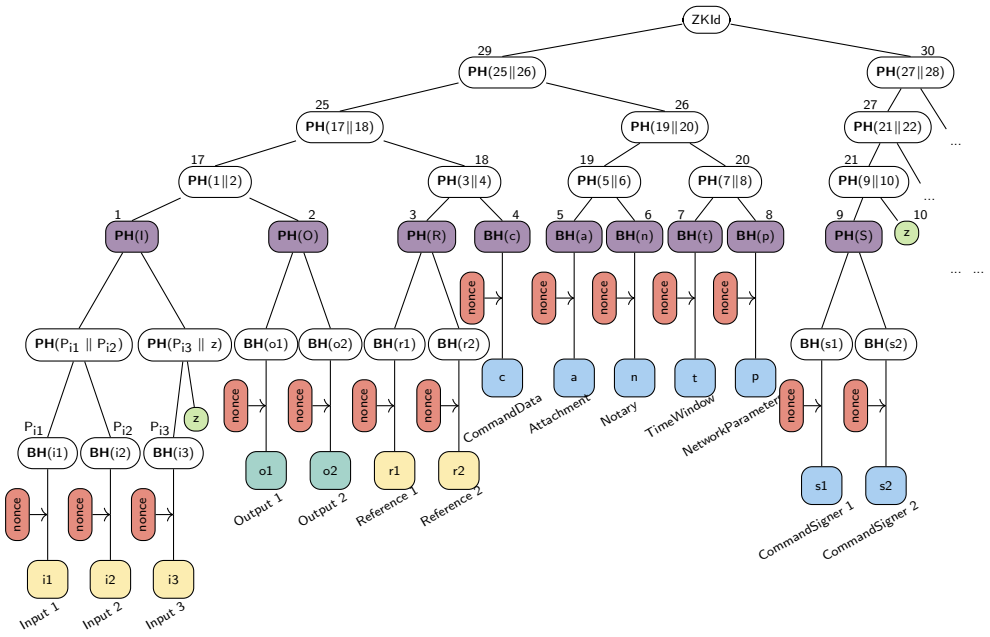

Figure 1: Merkle Tree representation of a transaction in ZKFlow.

In this document, we investigate alternative ZKP-friendly hash functions that can be used in ZKFlow to improve the performance of the protocol. We evaluate each hash function with respect to the following conditions:

- It should be more efficient than Blake2s and, favorably, Pedersen hash for the computations within the circuit.

- It should be more efficient than Pedersen hash and be as close as possible to Blake 2s in efficiency for the computations in plaintext (out of the circuit).
- It should not degrade the security guarantees of Blake2s and Pedersen hash.

In our evaluation, we consider the following hash functions:

- Pedersen & Sinsemilla
- Poseidon
- Rescue & Rescue-Prime
- MiMC
- Reinforced concrete.

In the rest of the document, we first give a brief description of each function and discuss their security guarantees with respect to collision and pre-image resistance. Secondly, we provide a benchmark of these hash functions in comparison to standard hash functions SHA256 and Blake2s for their performance in the ZKP circuit and in plaintext. Finally, we analyze the risk and impact of using any of these functions in the computation of component leaf hashes for ZKFlow in the existence of any security failures. Our main objective is to analyze how likely a collision or pre-image attack can impact the security in practice. We discuss for certain hash functions whether using fixed length input per component group reduces the collision probability. We also look at if Corda protocol provides mechanisms to prevent such attacks. We analyze each level of the Merkle tree to understand what a failure means in the corresponding level. For each level, we answer the following questions:

- What is the impact of an attack by collision for the corresponding input structure?
- Does Corda protocol provide a mechanism that prevents the attack?

2 ZKP-friendly Hash Functions

2.1 Pedersen Hash

Pedersen hash is an algebraic hash function whose security is based on the hardness of the discrete logarithm problem. In our protocol, we use the Zcash variant of the hash function that is optimized for efficient instantiation [HBHW21].

Pedersen hash processes a message in message blocks, where each message block is split into small sized chunks for injective mapping. In Zcash, the chunk size is 3. Therefore, prior to hashing, a message M is padded to a multiple of 3 bits, M', by appending 0 bits. For message blocks $M' = M_1||\cdots||M_n$, the hash digest is computed as

$$PH(D, M) = \sum_{i=1}^{n} \langle M_i \rangle I_i^D.$$

In the computation, the value D is the personalization parameter that is used in the generation of the generators I_i^D. The length of each message block M_i is $3c$ bits. The last block M_n might be shorter. c is the largest integer that assures the range of function $\langle M_i \rangle$ is in $\left[-\frac{r-1}{2}, \frac{r-1}{2}\right] - \{0\}$, where r is the group order. The function $\langle \cdot \rangle$ is computed on each chunk $m_j = [s_0^j, s_1^j, s_2^j]$ as

$$\langle M_i \rangle = \sum_{j=1}^{k_i} enc(m_j) 2^{4(j-1)}, \; enc(m_j) = (1 - 2s_2^j)(1 + s_0^j + 2s_1^j)$$

2.2 Sinsemilla Hash

Sinsemilla is an efficient version of Pedersen hash that is designed by the Zcash team for algebraic circuits that supports plookups, such as PLONK and Halo2 [HBHW21]. Its security guarantees are the same as Pedersen hash:

- the hash function is collision resistant for fixed length messages based on the hardness of discrete logarithm problem.
- it is not suitable to be used as a random oracle or PRF.

We do not provide details of how Sinsemilla hash function works since it is similar to Pedersen hash. We refer interested readers to Zcash protocol documentation for the details [HBHW21].

2.3 MiMC Hash

MiMC [AGR+16] is one of the earlier hash functions that is designed as a ZKP-friendly hash function. Different from the Pedersen hash, its design is based on block cipher design. The hash function is an instantiation of a block cipher with fixed round constants using sponge construction (see Definition 2.3). MiMC has a simpler design compared to classical symmetric cipher designs. It builds upon

- a permutation function $f(x) = x^3$,
- and subkey additions

in each round. It can operate both on F_p and F_{2^n}.

Sponge Construction

A sponge [BDPA08] is an alternative to compression functions that are used in encryption, authentication, hashing mechanisms. It builds upon a permutation or a fixed-length transformation and maps variable-length input to variable-length output. The sponge construction is build on the parameters

- f fixed-length internal permutation, operating on fixed number of b bits such that $b = r+c$,
- r bit rate, determines the throughput,
- c capacity, determines the security level [BDPA08].

Sponge operates in two phases: absorbing phase and squeezing phase. To perform sponge on an input string, first the input is cut into blocks of r bits and padded if necessary. Then,

- in the absorbing phase, the r-bit input blocks are XORed with the first r bits of the state, whose initial value is zero and the permutation f is applied on each XOR output;
- in the squeezing phase, the first r-bits of the state is returned as the output block such that between every block the permutation f is applied on the state.

Following is an illustration of the sponge construction as proposed in [BDPA08].

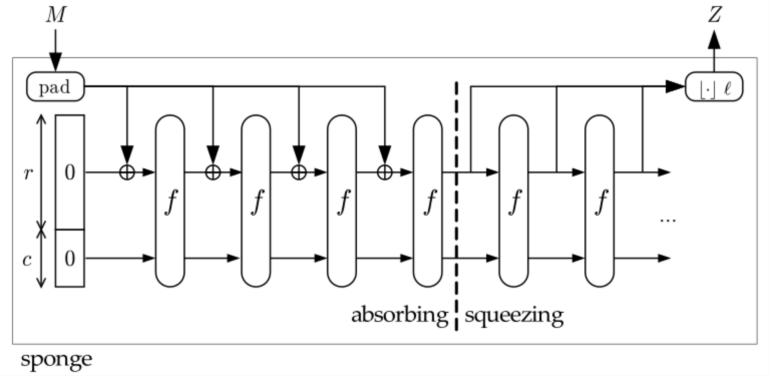

Figure 2 illustrates the rounds of MiMC as a block cipher. For the hash function, in the sponge construction the permutation f is set as $f(x) = x^3$. In hashing with MiMC, padding is required if the input string is not a multiple of the rate r.

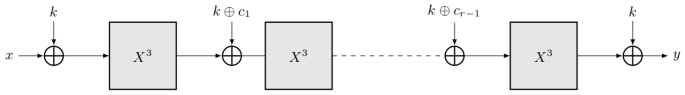

Figure 2: MiMC block cipher construction.

2.4 Poseidon Hash

Poseidon is a cryptographic permutation that is originally designed as a block cipher. The permutation can be used as a hash function by replacing the round keys of the cipher with fixed, independent round constants and applying the sponge construction[GKK+20].

Poseidon maps strings over the prime field F_p^* to fixed length strings over F_p. It is constructed by instantiation of a sponge function with a Poseidon permutation. Different from classical SPN (substitution-permutation network) designs, the Poseidon permutation is based on Hades design strategy [GLR+20]. Hades proposes to use different round functions within the same construction such that some rounds use full S-box layers and some uses partial S-box layers. The motivation for this design decision is that full S-box layers are expensive in ZKP-circuits, while partial layers can be performed more efficiently in circuit. However, using partial layers might degrade the resistance against algebraic attacks. In Poseidon, in total $2R_f + R_P$ rounds are applied within a permutation such that it starts with full rounds (R_f), followed by (R_P) partial rounds and ends with R_f full rounds. Each round contains the following operations:

- *AddRoundConstants* ($ARC(\cdot)$), where round constants are added to the state,
- *SubWords* (S), the S-box layer s.t. S-box$(x) = x^\alpha$, whose number differs between R_f and R_P rounds,
- *MixLayer* $M(\cdot)$, where a matrix multiplication with an MDS matrix (maximum distance separable) is performed[1].

Figure 3 shows an overview of Poseidon construction. The value of α, number of rounds and the round constants differs with respect to the field used, the security parameter and the proof system. We refer readers to the original paper for the details on the parameters [GKK+20]. Similar to MiMC, since sponge is used in hashing, if the message size is not a multiple of rate r, the message should be padded.

2.5 Rescue Hash

Rescue hash is a member of Marvellous design strategy, which aims to design ciphers efficient in arithmetic complexity [AAB+20]. It operates on field elements in F_p. The design of the hash function is similar to Poseidon with the use of S-boxes, MDS matrices, and building upon sponge construction. However, unlike Poseidon, it uses full rounds in every round, i.e., it utilizes SPN architecture. Each round of Rescue contains the following operations:

- *S-box layer*, where S-box$(x) = x^\alpha$ is applied to each element of the state,
- *Linear layer*, where a matrix multiplication with the MDS matrix is performed,
- *Constants injection*, where the round constants are added,
- *Inverse S-box layer*, where the inverse S-box function S-box$^{-1}(x) = x^{\alpha^{-1}}$ is applied,
- *Linear layer*,
- *Constants injection* [AAB+20].

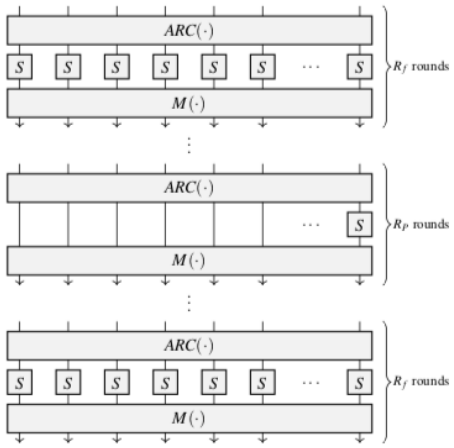

Figure 3: Poseidon hash function construction.

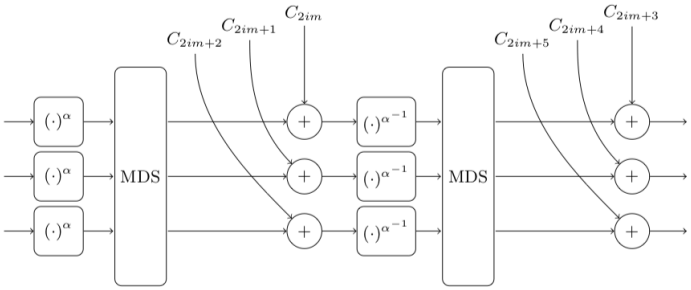

Figure 4: One round of Rescue hash construction.

Figure 4 illustrates the operations performed in a single round of Rescue hash.

Apart from Rescue, there exists several other hash functions in the Marvellous universe, which are Vision [AAB+20], Rescue-Prime [SAD20], Jarvis [AD18], and Friday [AD18]. Vision is the version of Rescue that works on binary fields, F_{2^n}, with similar design and security guarantees. Rescue-Prime is an optimization of Rescue hash. We included it in our benchmarks separately to observe its performance improvement. Jarvis (block cipher) and Friday (hash function) are the predecessors of Rescue and Vision. Unfortunately, both designs are vulnerable against algebraic attacks as shown in [ACG+19], thus, not included in this document.

2.6 Reinforced Concrete Hash

Reinforced Concrete (RC) [BGK+21] is the most recent proposal for the ZKP-friendly hash functions. It makes use of lookup tables in its design and, because of that, targets proof systems that supports lookup operations, Plookup [GW20]. It operates on fields elements in F_p.

The design of RC, illustrated in Figure 5, is similar to Poseidon. The hash function uses two types of rounds: external rounds with full S-box layers and internal rounds with partial S-box layers. Different from Poseidon, in the internal rounds, instead of x^{α}, a complex algebraic structure is used. The rationale behind this design is to protect the hash function from statistical attacks with the use of external rounds and from algebraic attacks with the complex function in the internal rounds [BGK+21].

In Figure 5, in the `CONCRETE` layer multiplication with the MDS matrix is performed. In the `BRICKS` layer, a nonlinear transformation with a permutation of degree d is performed. Finally, in `BARS` layer a special permutation is applied as described in [BGK+21].

[1] https://en.wikipedia.org/wiki/MDS_matrix

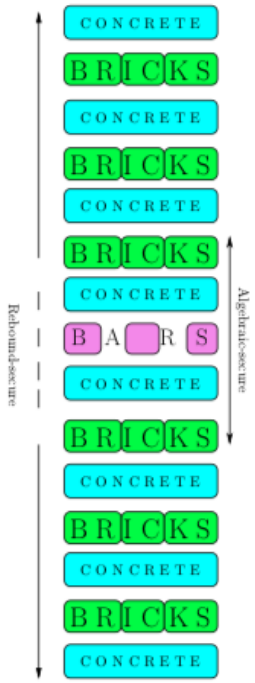

Figure 5: Reinforced Concrete hash construction.

3 Comparison of ZKP-friendly functions

In this section, we compare the ZKP-friendly hash functions with respect to their computational performance and security guarantees.

3.1 Performance Analysis

In the comparison of ZKP-friendly primitives with respect to their performance, we consider two performance measurements:

- *The performance of the hash function in ZKP circuit:* Since existing standard primitives are prohibitively expensive in arithmetic circuits, the need for ZKP-friendly primitives has risen. Therefore, in performance comparison, the hash function with lowest number of constraints is more favourable.
- *The performance of the hash function in plaintext:* Transaction validation in ZKFlow requires to compute the Merkle root of the transaction by the transaction creators. Therefore, being efficient within circuit is not sufficient. The hash function should also have a computational performance that is as good as standard primitives, such as SHA256, Blake2s. Therefore, we also compare each hash function with respect to their plaintext performance.

Table 1 provides an overview of computational performance of each hash function for hashing inputs of 512 bits as reported in [BGK+21]. The results show that Reinforced Concrete provides the fastest performance in plaintext. It also has the lowest number of constraints for the proof systems that uses plookup. Poseidon, on the other hand, provides the best performance in R1CS based systems while ranking as the second best in plaintext performance.

As a second experiment, we analyzed the cost of computing hash functions on large input sizes. For our experiment, we used the hash function implementations in `franklin-crypto` library [2]. The library does not include implementations in both proof systems (R1CS-based and Plookup-based) and some hash functions have limit on input sizes. Therefore, we performed our tests only on Pedersen, Poseidon, Rescue, and Rescue-Prime.

[2] https://github.com/matter-labs/franklin-crypto

	Performance		
	Plaintext (ns)	R1CS (# constraints)	Plookup (# constraints)
SHA256	366.5	27534	∼3000
Blake2s	219.5	21006	∼2000
Pedersen	39 807	869	–
Sinsemilla	131 460	–	670
MiMC	33 800	1326	1326
Poseidon	19 464	243	438
Rescue	415 230	288	364
Rescue-Prime	362 870	252	321
Reinforced Concrete	3 284	–	267

Table 1: The cost of hashing 512 bits (32 bytes) of data using different hash functions, retrieved from [BGK+21].

Table 2 shows the results of our experiments. The results are inline with the results of the first experiment. We observe that Poseidon has the best performance in plaintext and also in-circuit computation time. With respect to the number of constraints Rescue-Prime has a better performance, but its plaintext performance is much slower than Poseidon.

	Performance			
	Plaintext (ms)	In-circuit (ms)	R1CS (# constraints)	Plookup (# constraints)
Pedersen	149732 (generators) 191 (hash)	13020	216031	–
Poseidon	61	1462	–	44110
Rescue	2399	4730	–	83203
Rescue-Prime	820	1685	–	30449

Table 2: The cost of hashing 10KB of data using different hash functions.

3.2 Security analysis

In this section, we provide a comparison of the security claims for each ZKP-friendly hash function. Table 3 gives an overview of security claims in terms of collision resistance, preimage resistance, and second preimage resistance with respect to the output length of each hash function. To give a better insight in comparison, the security claims of SHA256 and Blake2s are also included in the table.

	Output length (bits)	Collision resistance	Preimage resistance	2^{nd} Preimage resistance
SHA256	256	128	256	$256 - \log_2 L^*$
Blake2s	256	128	256	256
Pedersen	256	128	128	128
Sinsemilla	256	128	128	128
MiMC	256	128	256	256
Poseidon	256	256	256	256
Rescue	252	128	128	128
Rescue-Prime	252	128	128	128
Reinforced Concrete	256	128	128	128

Table 3: The security of hash functions with respect to the output length.

There are several types of attacks that can target the claimed security guarantees of the proposed

hash functions. These attacks target permutation-based hash functions, i.e., MiMC, Poseidon, Rescue, and Reinforced Concrete. Since the security argument of Pedersen and Sinsemilla hashes are different, we discuss possible attack factors for these hash functions later in a separate subsection (Section 3.2.1). The internal permutation in a hash function can be attacked by

- **Statistical cryptanalysis** which exploits the undesirable probabilistic properties of ciphers. There are two well-known types of statistical attacks:
  - **Differential cryptanalysis** which aims to get information from a pair of ciphertext by observing particular differences in their corresponding plaintexts [BPW05].
  - **Linear cryptanalysis** which aims to find a linear relation between some bits of the plaintext, ciphertext and the unknown key [BPW05].
- **Algebraic cryptanalysis** which exploits algebraic properties of of ciphers. Algebraic attacks are more powerful for lightweight ciphers with simple algebraic structures. Thus, resistance against algebraic attacks is crucial for ZKP-friendly designs which requires low complexity for computational efficiency. Some well-known algebraic attacks are as follows:
  - **Gröbner basis attack** which tries to recover the secret key from a small number of plaintext/ciphertext pairs by computing Gröbner bases, which is a special structure that helps solving polynomial equations [BPW05].
  - **Interpolation attack** where an attacker tries to reconstruct the polynomial that describes the cipher by applying Lagrange interpolation on the plaintext/ciphertext pairs [AAB+20].
  - **Higher-Order Differential attack** considers the cipher as a multivariate polynomial. The attack is applied on a simplified version of the polynomial which is obtained by summing over some subsets. Then, the goal is to solve the subset equation to get the bits of the secret key.

We do not provide the details of how each attack works and how each hash function can provide security against above mentioned attacks. Instead, in the following we provide a brief summary about each hash function whether they have been attacked by any of these attacks:

- **MiMC:** As one of the earliest ZKP-friendly friendly hash functions, MiMC had a chance of thorough security investigation. Its block cipher variant (based on Feistel construction) and the binary field version was shown vulnerable to certain algebraic attacks [Bon19, EGL+20]. Furthermore, an extension of MiMC, GMiMC is shown to suffer from practical collision attacks [Bon19, BCD+20]. However, there are no known attack on the MiMC hash function to date. Due to its security guarantees, MiMC was considered one of the final candidates for the STARK-friendly Hash Function Challenge of Ethereum Foundation [BGL20].
- **Poseidon:** An earlier round reduced version of Poseidon hash [GKK+19] was shown to be vulnerable against algebraic attacks in its inner layers which results in a preimage attack [BCD+20]. The cryptanalysis result states that the security margin of Hades design strategy might be lower than the claimed security level in the existence of partial rounds. While using partial rounds can improve computational efficiency, it might degrade the security of the hash function. The designers of Poseidon provided an updated version of their hash function by addressing the possible attacks. Due to these attacks, Poseidon was not considered as a final candidate in STARK-friendly hash challenge. Furthermore, its use in Zcash protocol is limited to PRF [HBHW21].
- **Rescue:** To date there are no known attacks on Rescue hash function, apart from the attacks on its previous versions. Compared to Poseidon, the use of full rounds in Rescue's design and its similarity to classical SPN networks makes it secure against known cryptanalytic techniques [BCD+20]. For this reason, it was selected as the finalist of STARK-friendly hash challenge [BGL20].
- **Reinforced concrete:** As the most recent of the considered hash functions, Reinforced Concrete has not been through a wide security investigation yet. The designers of the hash function show that none of the aforementioned attacks are applicable on RC. However, there is no external cryptanalysis effort that supports this claim yet. We are aware of a recent hash challenge to

investigate the security of RC along with other hash functions [3]. Therefore, currently, we cannot make any claim on the security of Reinforced Concrete hash.

3.2.1 Pedersen & Sinsemilla Hash Security Analysis

Since Pedersen and Sinsemilla hashes have a different structure than the other hash functions, we analyse their security separately in the following.

Collision resistance

- **Collision due to padding** Pedersen hash requires each message to be a multiple of 3. Therefore, the messages are padded with zero bits if they do not satisfy this requirement. Unfortunately, this padding causes collisions for the messages in the same window. Explicitly, given message $|M| \equiv 1 \mod 3$, the hash digests for messages $PH(M)$, $PH(M||0)$, and $PH(M||0||0)$ are equal since they are all padded to $M||0||0$. This security issue can be avoided by prefixing each message with its message length as $PH(\ell_M||M)$[4].
- **Collision in prefixed messages** We discussed that adding the message length as a prefix to the preimage prevents collision risks due to padding. However, we might still observe collisions, if the outputs of $\langle M_1 \rangle$ and $\langle M_2 \rangle$ are the same for the different messages M_1 and M_2, i.e.

$$\sum \langle M_i^1 \rangle I_i^D = \sum \langle M_i^2 \rangle I_i'^D.$$

  As proved in [HBHW21], the function $\langle \cdot \rangle$ is injective, therefore the output is distinct for each distinct message. Furthermore, the message blocks constructed in a certain format that the output cannot exceed the group order r. Under these conditions, finding messages with the same hash digests is equivalent to finding a discrete logarithm relation between the generators, which is not feasible with the chosen security parameters.

Length extension attacks Pedersen hash is a homomorphic hash function. Therefore, it is possible to compute a hash digest from two different hash digest as

$$PH^{I_1}(M_1) + PH^{I_2}(M_2) = PH^{I_1||I_2}(M_1||M_2).$$

A malicious adversary can exploit this property to perform a length extension attack, if there are some unused generators for shorter messages to compute valid hash digests. We can prevent such an attack using message size prefixing. Thus, an adversary that wants to compute the concatenation of two messages will get the digest

$$PH^{I_1}(\ell_{M_1}||M_1) + PH^{I_2}(\ell_{M_2}||M_2) = PH^{I_1||I_2}(\ell_{M_1}||M_1||\ell_{M_2}||M_2),$$

instead of

$$PH^{I_1||I_2}(\ell_{M_1||M_2}||M_1||M_2).$$

Alternatively, an active adversary can alter hash computation by removing the prefix message length to perform the following homomorphic computation:

$$PH^{I_1}(\ell_{M_1}||M_1) + PH^{I_2}(M_2) = PH^{I_1||I_2}(\ell_{M_1}||M_1||M_2).$$

An attacker might attempt to use the aforementioned homomorphic computations to attack ZKFlow in following scenarios:

- **To replace a component leaf hash with a malicious hash value:** An attacker might alter the leaf hash of a component $PH(M)$ with a malicious digest $PH(M||M^*)$. To be able to

[3] https://www.zkhashbounties.info
[4] https://forum.zcashcommunity.com/t/pedersen-hash-collision-resistance/33586/2

succeed the attacker should find a collision such that $PH(M) = PH(M||M^*)$ due to Merkle tree validation.

$$PH(M) = \sum \langle M_i \rangle I_i^D$$
$$PH(M||M^*) = \sum \langle M_i \rangle I_i^D + \sum \langle M_j^* \rangle I_j^D.$$

However, this is not a trivial computation. The output of function $\langle \cdot \rangle$ is non-zero. Thus, the attacker should find a message whose homomorphic addition along with a specific generator results in the same hash value. This is equivalent to finding discrete logarithm.

- **To create a new component group hash from the old one:** If an attacker knows the structure of preimage, they can alter certain parts of the message or append new values by using homomorphic property of the Pedersen hash. For example, given the hash of a component leaf hash in a simple form as

$$PH(M) = \ell \cdot I_1 + \text{nonce} \cdot I_2 + M \cdot I_3,$$

an attacker can alter the hash to a message of same size as

$$PH(M + M') = \ell \cdot I_1 + \text{nonce} \cdot I_2 + M \cdot I_3 + M' \cdot I_3.$$

The attacker cannot use this new value in the current transaction since the Merkle root validation is going to fail.

The attacker might attempt to use this value to create a new transaction. Thus, by keeping the structure $\ell \cdot I_1 + \text{nonce} \cdot I_2$ fixed, attacker can alter the computation $M \cdot I_3 + M' \cdot I_3$ for each component value. Despite the attacker might create a valid Merkle tree structure with this attack, they cannot convince an honest verifier for the validity of structure since the corresponding ZKP for the transaction cannot be validated without knowing the privacy salt. The ZKP requires to validate the computation of nonce values from the privacy salt. Without having access to the privacy salt, an attacker cannot replay the computation of nonce as a Blake2s digest from the privacy salt.

Alternatively, an attacker might try to remove the nonce from the hash digest using the homomorphic property of the Pedersen hash to apply a preimage attack on the message. If the message length is small, then attacker might be able to find the preimage. For the public components of the transaction this do not pose an extra risk since preimage is already publicly available. For instance, for the input component group both nonce and input `StateRef` values are already available. It might pose a risk for output states but, to succeed, the attacker should first find the right nonce value for the output states which is a private value in ZKFlow. Knowing the nonce values of other component group elements does not give an advantage to the attacker since each nonce value is computed randomly using Blake2s hash function. Therefore, the preimage attack cannot succeed.

Preimage resistance The preimage search cost of Pedersen Hash is $2^{\frac{\ell}{2}}$, where ℓ is the length of the message in bits[5]. In ZKFlow, the preimage of component leaf hashes have the following structure

$$PH(\ell||\text{nonce}||M),$$

where ℓ is a 32-bit integer for the message length, nonce is a 256-bit hash digest. Depending on the component group, the message size might vary. In our computation, we take the shortest message size in ZKFlow which is a byte. Then, the total message length is at least 296 bits. The preimage search cost will be at least

$$2^{\frac{296}{2}} = 2^{148},$$

which is secure with the current computational limits (112 bits or 128 bits for long term.[6])

[5] https://t.me/real_crypt/60
[6] https://www.keylength.com/en/4/

- **Multi target preimage resistance** The multi target preimage search cost of Pedersen Hash is $2^{\frac{\ell}{2}-\frac{k}{2}}$ for 2^k pre-images with ℓ-bit message size. In ZKFlow, given the 296-bit message length for shortest message, achieving this attack requires at least

$$2^{\frac{296}{2}-\frac{k}{2}} > 2^{128},$$
$$148 - \frac{k}{2} > 128,$$
$$40 > k,$$

2^{40} preimages, which makes the attack possible for the chosen message size. In Table 4, we look at the multi target preimage search cost of different component element types in ZKFlow.

	element bits	preimage bits	min #preimages for 112-bit	min #preimages for 128-bit
boolean	8	296	2^{72}	2^{40}
integer	32	320	2^{96}	2^{64}
hash digest	256	544	2^{320}	2^{288}
public key	352	640	2^{416}	2^{384}

Table 4: The multi target preimage search cost for several component element types in ZKFlow.

The numbers in Table 4 show that for the commonly used types in ZKFlow, i.e., hash digest and public key, the search cost for multi target preimaga attacks are not feasible. For the types that have lower search cost, an attack can be mitigated by padding the messages with random values to a minimum length, where an attack is infeasible.

4 Analysis on each component group element

In this section, we look at the impact of a failure in the security of a ZKP-friendly hash for each component group individually. Since Pedersen hash guarantees collision resistance for fixed length messages, when necessary we discuss how the security is affected with fixed length or arbitrary length messages when Pedersen hash is used. The same conditions applies for Sinsemilla hash. For all other hash functions discussed, padding is required when the input length is not a multiple of rate value r in sponge construction. However, it does not affect the security guarantees of the hash functions.

When the size of the element is fixed, as in the current design, an adversary cannot attack the ZKFlow protocol. Since Pedersen hash is collision-resistant for fixed size messages, it is not feasible to find other messages with the same length that have the same hash value as the original message. An attacker might be able to generate the same hash value from a different length message. However, due to fixed size constraint in the protocol, this message value will not be accepted by the protocol. Therefore, in the rest, we only look at risks in the existence of arbitrary element sizes.

4.1 Input & Reference Component Groups

Element type: `StateRef` = Merkle root + index = HashDigest + Integer

4.1.1 Arbitrary element size

- **What is the impact?** The attacker might attempt to replace the valid `StateRef` value with the arbitrary message that has the same hash digest as the valid input. Thus, the attacker is able to break the transaction backchain of the attacked input element.
- **Does Corda prevent it?** The validation of the transaction requires validation of the backchain. If the backchain validation does not succeed, then the transaction will not be validated and discarded. Therefore, the attack does not succeed. Furthermore, `StateRef` values are hash digests of an expected fixed length, and the protocol expects to operate on the predetermined length input size. Therefore, in practice, the attacker cannot change the size of `StateRef` to an arbitrary value.

4.2 Output Component Group

Element type: `TransactionState` = Variable content/ length based on the application.

- **What is the impact?** The attacker can replace the valid `TransactionState` value with an arbitrary message that has the same hash digest as the valid input. The attacker can implement a different contract state that gives the same hash digest.
- **Does Corda prevent it?** Despite the attacker's ability to find a collision, this attempt will fail due to Corda's validation mechanism. In the Corda protocol, one of the attachments to every transaction is the contract code jar. The hash of this attachment, is a SHA256 digest, i.e. `AttachmentId`. This is added to the Merkle tree as an attachment. When a transaction is verified, the verifier loads the contract code for the contract's `AttachmentId` and verifies the transaction contents using that contract code's logic.

  It is not trivial to provide malicious contract code to a verifier, since its `AttachmentId` is SHA256, which is collision resistant for arbitrary length messages. The contract code will expect TransactionState contents to be of a certain type for the command component found in the transaction. If the TransactionState is of a different type, contract verification will fail. This leaves the attacker with one option: to find a collision for a TransactionState of the same type, but with changed values.

4.3 Command Component Group

Element type: `CommandData` = Variable content based on application

- **What is the impact?** The attacker can replace the valid `CommandData` value with the arbitrary message or another valid command value that has the same hash digest as the valid command.
- **Does Corda prevent it?** Corda provides two mechanisms to prevent this attack. The first one is during the validation of transaction structure. In the validation of `LedgerTransaction`, each command is matched with its signer keys. Therefore, when an arbitrary value is assigned as the `CommandData`, the check on the signers and `CommandData` value is going to fail and attack cannot succeed.

  On the other hand, the `CommandData` value used by the attacker can be a valid one and it might have the same signers list as the original one. In that case, the contract rules validation can help to prevent the attack. If the transaction structure does not meet the contract rules of the chosen malicious `CommandData` value, then contract rules validation will fail and attack cannot succeed.

4.4 Attachment Component Group

Element type: `AttachmentId` = SHA256(Attachment)

- **What is the impact?** In the case of attachments, the arbitrary message length does not affect the security. Because the challenge for the attacker is to find a collision in SHA256 which is not possible currently. This attack is not feasible due to the collision-resistance guarantee of the SHA256 hash.
- **Does Corda prevent it?** A possible attack is prevented thanks to Corda's design of including attachment ids to the transaction instead of attachment content.

4.5 Notary Component Group

Element type: `Party` = CordaX500Name || PubKey

- **What is the impact?** An attacker can modify the content of CordaX500Name and find a fake value that has the same has digest as the original value. Similarly, the attacker might also find arbitrary length values for PubKey. The attacker might attempt to introduce a fake notary to the network. Alternatively, the attacker might destroy the PubKey of the notary by changing its content to an arbitrary value.

- **Does Corda prevent it?** Corda maintains a list of `legalIdentities`, where each party's validity is checked during transaction validation. Therefore, an arbitrary CordaX500Name cannot be validated. An arbitrary PubKey value cannot be used in validation since there would not be a corresponding private key for it.
- If the two Party elements have the same hash digest, is there a way to validate the identities?

4.6 TimeWindow Component Group

Element type: `TimeWindow` = Instant (12-byte date value)

- **Is there a feasible attack?**
- **What is the impact?** An attacker might alter the value of `TimeWindow` value.
- **Does Corda prevent it?** A fake `TimeWindow` value will fail in transaction validation. Therefore, the attack does not succeed.

4.7 Network Parameters Hash Component Group

Element type: `NetworkParam` = SHA256(NetworkParameters)

- **What is the impact?** Similar to the `AttachmentId`, the preimage of the network parameters hash component elements is SHA256 digest. Therefore, finding a collision is only possible if collision in SHA256 is feasible. There is no known vulnerability around collision-resistance of SHA256. Therefore, an attack would not be successful.
- **Does Corda prevent it?** A possible attack is prevented thanks to Corda's design of including network parameters hash digest to the transaction instead of its full content.

4.8 Signers Component Group

Element type: `Signer` = PubKey

- **What is the impact?** An attacker might block the usage of PubKey.
- **Does Corda prevent it?** The fake value cannot be used in the validation of signatures since there is no corresponding valid private key value for it. Therefore, the validation will fail and transaction proposal will not be accepted.

4.9 Cross Component Group Collision

Apart from within component group collisions, it is also possible to observe collisions cross component groups. For instance, the component leaf hash of an Input component group element might have the same digest value with the component leaf hash of the Notary group element. However, the probability of this collision is very low since in the computation of leaf hashes unique random nonce values are used.

If anyhow a collision is found, in the existence of fixed length message sizes, an attacker cannot swap the preimages of two colliding hashes since the protocol will fail. Even in the existence of arbitrary length messages, the validation mechanisms of Corda will abort transaction validation as explained above.

4.10 Analysis on component group hashes

On the level of component group hashes, the following possibilities for collision do not threaten the correctness of the protocol:

1. Two component group hashes might have the same hash value
2. A component group hash might have the same hash value with another component group leaf hash.

Fixed or variable length message size in component groups does not affect the collision likelihood since in both cases it is possible to have collisions. Both cases do not threaten the protocol since we do not perform validation on component group hash level.

4.11 Analysis on the upper levels of Merkle tree

On the upper levels of the tree, a collision among one of the internal nodes of the tree and one of component leaf hashes can be observed. However, similar to the previous case, this does not have an impact on the correctness of the protocol.

5 Conclusion

In this document, we analyzed alternative hash functions to be used in the ZKFlow protocol that can perform more efficiently in ZKP circuits. In our analysis, we considered two main aspects: security guarantees of the hash function and performance of the hash function in the circuit and plaintext. Our analysis shows that Rescue hash provides the best security guarantees with no known successful attack on it to date. However, concerning performance, Rescue is slower than other candidates both in circuit and in plaintext. Poseidon hash, on the other hand, provides a better performance both in circuit and in plaintext, although there are some existing attacks on its reduced round variants. Among the candidates that were evaluated, Reinforced Concrete provides the best performance both in plaintext and in-circuit (when plookup is considered) and it is claimed to be resistant against the known attacks. However, due to its immaturity, we are hesitant to make any strong claim in its immediate usage in ZKFlow.

References

[AAB+20] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Trans. Symmetric Cryptol.*, 2020(3):1–45, 2020.

[ACG+19] Martin R. Albrecht, Carlos Cid, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, and Markus Schofnegger. Algebraic cryptanalysis of stark-friendly designs: Application to marvellous and mimc. In *ASIACRYPT (3)*, volume 11923 of *Lecture Notes in Computer Science*, pages 371–397. Springer, 2019.

[AD18] Tomer Ashur and Siemen Dhooghe. Marvellous: a stark-friendly family of cryptographic primitives. *IACR Cryptol. ePrint Arch.*, page 1098, 2018.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. *IACR Cryptol. ePrint Arch.*, page 492, 2016.

[BCD+20] Tim Beyne, Anne Canteaut, Itai Dinur, Maria Eichlseder, Gregor Leander, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Yu Sasaki, Yosuke Todo, and Friedrich Wiemer. Out of oddity - new cryptanalytic techniques against symmetric primitives optimized for integrity proof systems. In *CRYPTO (3)*, volume 12172 of *Lecture Notes in Computer Science*, pages 299–328. Springer, 2020.

[BDPA08] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[BGK+21] Mario Barbara, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Reinforced concrete: Fast hash function for zero knowledge proofs and verifiable computation. *IACR Cryptol. ePrint Arch.*, page 1038, 2021.

[BGL20] Eli Ben-Sasson, Lior Goldberg, and David Levit. STARK friendly hash - survey and recommendation. *IACR Cryptol. ePrint Arch.*, page 948, 2020.

[Bon19] Xavier Bonnetain. Collisions on feistel-mimc and univariate gmimc. *IACR Cryptol. ePrint Arch.*, page 951, 2019.

[BPW05] Johannes Buchmann, Andrei Pychkine, and Ralf-Philipp Weinmann. Block ciphers sensitive to groebner basis attacks. *IACR Cryptol. ePrint Arch.*, page 200, 2005.

[EGL+20] Maria Eichlseder, Lorenzo Grassi, Reinhard Lüftenegger, Morten Øygarden, Christian Rechberger, Markus Schofnegger, and Qingju Wang. An algebraic attack on ciphers with low-degree round functions: Application to full mimc. In *ASIACRYPT (1)*, volume 12491 of *Lecture Notes in Computer Science*, pages 477–506. Springer, 2020.

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. *IACR Cryptol. ePrint Arch.*, page 458, 2019.

[GKK+20] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. *IACR Cryptol. ePrint Arch.*, page 458, 2020.

[GLR+20] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a generalization of substitution-permutation networks: The HADES design strategy. In *EUROCRYPT (2)*, volume 12106 of *Lecture Notes in Computer Science*, pages 674–704. Springer, 2020.

[GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.*, page 315, 2020.

[HBHW21] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. https://github.com/zcash/zips/blob/master/protocol/protocol.pdf, September 2021.

[SAD20] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-prime: a standard specification (sok). *IACR Cryptol. ePrint Arch.*, page 1143, 2020.