Pedersen Hash Security Analysis for ZKFlow

April 2021

1 Introduction

Pedersen is a secure and efficient hash function based on elliptic curve cryptography. Its efficiency in arithmetic circuits makes them a favourable option to use in ZKP schemes compared to other standard hashing algorithms. The hash function guarantees collision resistance for fixed length messages for a chosen personalization input [1]. Therefore, for variable-length inputs with the same personaliztion value, it is not collision resistant. Furthermore, it is not suitable as a PRF due to its visible structure. Its security guarantees makes it suitable for Merkle tree constructions, where existence of collision resistance is adequate to construct the Merkle tree.

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 than Pedersen hash (21K constraints vs 1.3K constraints), which impacts the performance of ZKFlow significantly.

In this document, we first investigate security guarantees of Pedersen hash with respect to its collision-resistance, preimage resistance and second preimage resistance. We discuss how desired security guarantees can be achieved in ZKFlow with prefixing techniques and sufficiently large input sizes.

In the second part of the document, we analyze the risk and impact of using Pedersen hash in the computation of component leaf hashes in the existence of collisions. Our main objective is to analyze how likely collisions can impact the security in practice. We discuss 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 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?

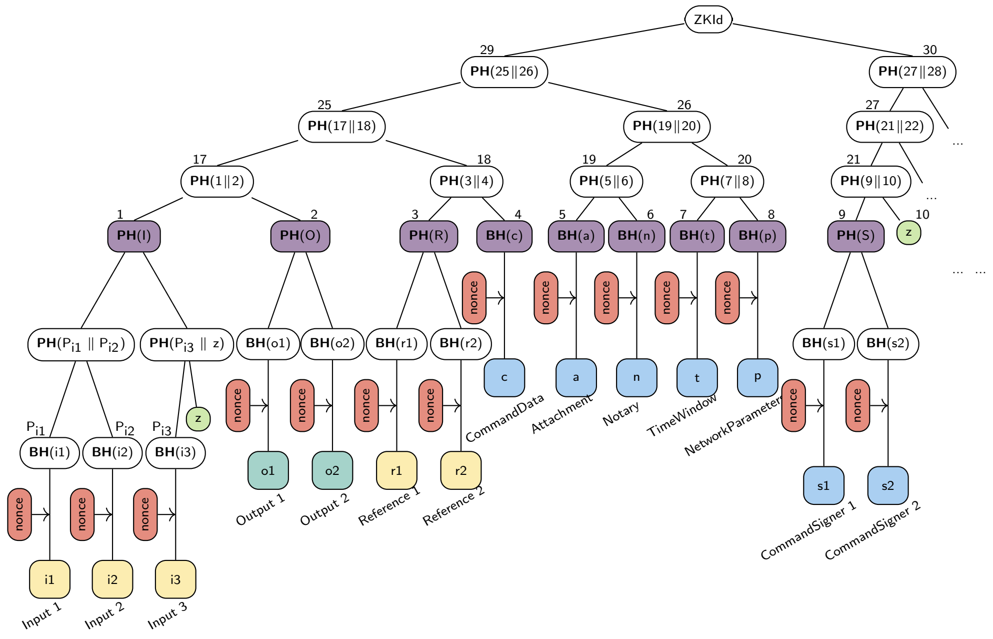
Figure 1: Merkle Tree representation of a transaction in ZKFlow.

- Does Corda protocol provide a mechanism that prevents the attack?

2 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 [1].

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

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

In the following, we analyze the collision resistance and preimage resistance of the Pedersen hash for the message format used in the ZKFlow protocol.

2.1 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 $PH(\ell_M||M)$[1]. .

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 [1], 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.

2.2 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

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

[1] https://forum.zcashcommunity.com/t/pedersen-hash-collision-resistance/33586/2

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

$$\begin{aligned} PH(M) &= \sum \langle M_i \rangle I_i^D \\ PH(M||M^*) &= \sum \langle M_i \rangle I_i^D + \sum \langle M_j^* \rangle I_j^D. \end{aligned}$$

  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.

2.3 Preimage resistance

The preimage search cost of Pedersen Hash is $2^{\frac{\ell}{2}}$, where ℓ is the length of the message in bits[2]. 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.[3])

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

$$\begin{aligned}&2^{\frac{296}{2}-\frac{k}{2}} > 2^{128},\\&148 - \frac{k}{2} > 128,\\&40 > k,\end{aligned}$$

[2]`https://t.me/real_crypt/60`
[3]`https://www.keylength.com/en/4/`

2^{40} preimages, which makes the attack possible for the chosen message size. In Table 1, 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 1: The multi target preimage search cost for several component element types in ZKFlow.

The numbers in Table 1 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.

3 Analysis on each component group element

In this section, we look at the impact of a failure in the security of Pedersen hash for each component group individually. 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.

3.1 Input & Reference Component Groups

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

3.1.1 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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?

3.6 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.

3.7 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.

3.8 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.

3.9 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 Pedersen 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.

3.10 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.

3.11 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.

References

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