



# SC2SC Communication Protocol - Cryptography and Implementation Review

Zen Blockchain Foundation  
Version 1.0 – May 25, 2023

**Prepared By**  
Elena Bakos Lang  
Paul Bottinelli  
Aleksandar Kircanski

**Prepared For**  
Giacomo Gussoni  
Daniele Di Benedetto

# 1 Executive Summary

---

## Synopsis

In Spring 2023, Horizen Labs engaged NCC Group to conduct a security assessment of the Sidechain-to-Sidechain (SC2SC) communication protocol. The SC2SC communication protocol provides a way for multiple sidechains in the Horizen ecosystem to communicate with one another, by leveraging the sidechain to mainchain communication of the Zendoo Sidechain system. Three consultants performed the review, over a total of 10 person-days.

## Scope

NCC Group's evaluation included:

- The pull request implementing the circuit for SC2SC Verification <https://github.com/HorizenOfficial/zendoo-sc-cryptolib/pull/110>, and the corresponding [commit on branch sc2sc](#)
- [ZenIP-42205](#), the Horizen improvement proposal describing the required changes
- An internal [Design Document](#) providing additional details into the construction of the system

Additionally, an academic paper describing the high-level communication protocol <https://arxiv.org/pdf/2209.03907.pdf> was provided to facilitate the consultant's understanding of the system.

## Limitations

While the documentation provided described the entire SC2SC communication protocol, the code reviewed only covered the zero knowledge circuit and associated APIs. The sidechain functions implementing the remainder of the SC2SC protocol are also crucial to the overall security of the system, and were out of scope.

## Key Findings

Code and documentation review revealed two issues:

- **Incorrect Redeem Message Check**, in which a small design issue in the validation of *Redeem Messages* could allow users to redeem a message arbitrarily many times, thereby minting infinite tokens on the receiving sidechain, for example.
- **Unspecified Division of Responsibility Between SDK and Circuit**, in which security-critical checks ensuring the correct functioning and redeeming of messages could be missed as a result of the unclear division of responsibility between SDK and the arithmetic circuit itself.

Some additional informational notes are collected in [Engagement Notes](#).

*After retesting*, NCC Group found that the majority of findings have been addressed. In particular, one (1) finding has been marked as *Fixed*, and one (1) informational finding was marked as *Partially Fixed*. Additionally, NCC Group noted that Horizen Labs addressed many of the comments presented in the informational notes section. The findings and informational section have been updated with details of the changes made by Horizen Labs.

## Strategic Recommendations

Ensure that the design documents clearly delineate between the information verified within the cryptographic proof, and the information that must be validated by the redeeming sidechain. In particular, consider including a discussion of security-critical checks to be performed by the redeeming sidechain within [ZenIP-42205](#) or other developer-facing documents.

## 2 Dashboard

### Target Data

<b>Name</b>	SC2SC Communication Protocol
<b>Type</b>	Cryptographic Library
<b>Platforms</b>	Rust, Java
<b>Environment</b>	Local Instance


### Engagement Data

<b>Type</b>	Cryptography Implementation Review
<b>Method</b>	Source Code Security Review
<b>Dates</b>	2023-05-08 to 2023-05-12
<b>Consultants</b>	3
<b>Level of Effort</b>	10 person-days

### Finding Breakdown

Critical issues	0
High issues	1 
Medium issues	0
Low issues	0
Informational issues	1 
<b>Total issues</b>	<b>2</b>

### Category Breakdown

Cryptography	1 
Data Validation	1 

 Critical     High     Medium     Low     Informational



### 3 Table of Findings

---

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Incorrect Redeem Message Check	Fixed	2CR	High
Unspecified Division of Responsibility Between SDK and Circuit	Partially Fixed	D67	Info



## 4 Finding Details

High

### Incorrect Redeem Message Check

**Overall Risk** High  
**Impact** High  
**Exploitability** High

**Finding ID** NCC-E008290-2CR  
**Component** SC2SC  
**Category** Cryptography  
**Status** Fixed

#### Impact

A small design issue in the validation of *Redeem Messages* could allow users to redeem a message arbitrarily many times, thereby minting infinite tokens on the receiving sidechain, for example.

#### Description

First, let us summarize the message issuance/redeeming use-case (glancing over some of the more complex details). When a user issues a Cross-chain Message from a sidechain, that message eventually gets included in a message Merkle tree. The root of that Merkle Tree goes into a Certificate emitted by the sidechain and the hash of that Certificate is included in a Merkle tree on the Mainchain. The root of that Mainchain tree is included in a block. To redeem that message, the user posts a Redeem Cross-chain Message in the receiving sidechain, where a number of checks are performed to ensure that the correct message was redeemed. These checks include ensuring that the message was indeed part of the correct Merkle tree, and that the certificate was correctly included in the Sidechain commitment tree, on the Mainchain.

In the design document<sup>1</sup>, the section *CrossChainRedeemMsg validation* discusses the different steps to be taken to validate Redeem Messages, which we copy below, for reference.

```
The validation of the message will require several checks:
- CrossChainRedeemMsg.message.receivingScId = current sidechain
- hash(CrossChainRedeemMsg) has not already been redeemed
- CrossChainRedeemMsg.sc_commitment_tree_root and
↳ CrossChainRedeemMsg.next_sc_commitment_tree_root have been seen before
- CrossChainRedeemMsg.proof is valid (requires zk circuit invocation)
```

The check highlighted is meant to prevent redeeming the same message multiple times. To do so, a Sidechain maintains some data structure keeping track of which messages were previously redeemed. To check whether a message can be redeemed, that validation step highlighted above hashes the Redeem Message and ensures that it hasn't already been seen. The implementation of that procedure is outside the scope of this review, since this pertains to the SDK.

However, in this context, the Redeem Message is tied to the epoch in which a user is trying to redeem the message. Thus, redeeming the message in epoch N and N+1 will produce two different `CrossChainRedeemMsg`, and thus their hash will be different. Hence, a malicious user could redeem a `CrossChainMsg` multiple times, through several epochs.

1. <https://docs.google.com/document/d/1JEHWy3FjQwHAG2oQuy8wj3VxpdHDxz1PGFbm8Zt-B2s/edit#heading=h.3ucor72iy7nn>



---

## Recommendation

Replace the line highlighted above with something along the lines of the following.

```
- hash(CrossChainMsg) has not already been redeemed
```

Ensure this check is correctly performed by the SDK, and well-documented.

## Location

[SC2SC Design doc](#)

## Retest Results

**2023-05-25 – Fixed**

The *CrossChainRedeemMsg validation* section of the design document has been updated to require the following check, as per the recommendation.

```
- hash(CrossChainMsg) has not already been redeemed
```

This finding is thus considered *fixed*.



# Unspecified Division of Responsibility Between SDK and Circuit

**Overall Risk** Informational  
**Impact** High  
**Exploitability** Undetermined

**Finding ID** NCC-E008290-D67  
**Component** SC2SC  
**Category** Data Validation  
**Status** Partially Fixed

## Impact

The SC2SC design document specifies a number of security-critical checks ensuring the correct functioning and redeeming of messages. However, it is sometimes unclear whether these checks should be performed by the SDK or the arithmetic circuit itself. Some security-critical checks could be missed as a result.

## Description

The Cross-Sidechain Communication Protocol (SC2SC) is a fairly complicated process in which a user issues a message on a given sidechain, and later on redeems that message on another sidechain. In order to ensure that the redeem message is valid, it is accompanied by a zero-knowledge proof, proving that the original message was correctly submitted and committed to by both the sidechain and the mainchain.

However, the process by which a redeem message is validated and the different verification steps that should be undertaken to ensure of its authenticity are currently unclear; some checks are expected to be performed by the arithmetic circuit itself, while others should be done by the higher-level SDK. This distinction is not described in the current system design (i.e., in the current SC2SC design document<sup>2</sup>), where some steps are simply skipped.

For example, in the current protocol design, nothing ties a given message issuer to the cryptographic proof required to redeem that message. Thus, any user could, in theory, redeem another user's message. The Horizen Labs team indicated that such checks are expected to be performed in higher-level protocols and by the SDK itself, and not by the circuit handling proof verification. This should be clearly indicated somewhere, preferably in a document that sidechain developers will have easy access to.

Another example of this lack of clear distinction can be observed in the epoch checks enforcing certain messages have a certain maturity. The SC2SC design document states:

The process will first check that the CrossChainMsg has previously appeared, and it is not too recent: if we are at epoch N, we can redeem only messages appeared from epoch 0 to epoch N-2

This check again is to be performed by the SDK. In contrast, the circuit checks that the two certificates provided as witness to the proof verification are of continuous epochs, in the function `enforce_contiguos_epochs()`.

While a valid proof for a message in epoch X should not be able to be generated without epoch X+1 having concluded, the reasoning behind this assumption is not very clearly

2. <https://docs.google.com/document/d/1JEHWy3FjQwHAG2oQuy8wj3VxpdHDxz1PGFbm8Zt-B2s/edit#heading=h.3ucor72iy7nn>



---

presented in the design document. This document should make it clear where this check occurs, and how this property can hold in the system overall if it is only checked implicitly.

Additionally, note that [Zenip-42205.md](#) also does not clearly define roles and responsibilities with respect to the different validation steps to perform in the SC2SC protocol. In particular, the security-critical validation checks highlighted in the *CrossChainRedeemMsg* section of the design document are not discussed in ZenIP-42205. Since this document is likely to be consumed by a wider audience, it would be advisable to complement it with such discussion.

## Recommendation

Clearly document whether the circuit or the SDK is expected to perform the validation steps described in the SC2SC design document, both in the design document and in any document intended to be consumed by developers, such as [zenip-42205.md](#).

## Location

SC2SC Design doc

## Retest Results

### 2023-05-25 – Partially Fixed

The design document has been updated to include a clear division of the checks required from the SDK and the checks required from the circuit side. In the updated version, the checks listed in the *CrossChainRedeemMsg validation* section are now organized into *SDK side*, *Circuit side* and *Other checks* subsections, clearly identifying required checks from the SDK side, circuit side, and a brief discussion of possible optional checks to be performed by the application layer, respectively. In particular, a note about possible “restrictions on the user who can redeem a message” has been added in the *Other checks* section.

However, no clarification was observed regarding the implicit check of the following statement within the proof:

The process will first check that the *CrossChainMsg* has previously appeared, and it is not too recent: if we are at epoch N, we can redeem only messages appeared from epoch 0 to epoch N-2

Additionally, ZenIP-42205 has not been updated. As such, this finding is marked as *partially fixed*.



## 5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

### Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

### Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

### Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

### Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	



Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
<b>Low</b>	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

## Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
<b>Access Controls</b>	Related to authorization of users, and assessment of rights.
<b>Auditing and Logging</b>	Related to auditing of actions, or logging of problems.
<b>Authentication</b>	Related to the identification of users.
<b>Configuration</b>	Related to security configurations of servers, devices, or software.
<b>Cryptography</b>	Related to mathematical protections for data.
<b>Data Exposure</b>	Related to unintended exposure of sensitive information.
<b>Data Validation</b>	Related to improper reliance on the structure or values of data.
<b>Denial of Service</b>	Related to causing system failure.
<b>Error Reporting</b>	Related to the reporting of error conditions in a secure fashion.
<b>Patching</b>	Related to keeping software up to date.
<b>Session Management</b>	Related to the identification of authenticated users.
<b>Timing</b>	Related to race conditions, locking, or order of operations.



## 6 Engagement Notes

This informal section contains notes and observations generated during the project. There are no security issues that are not already reported in the preceding findings, but the following content may be useful for discussion purposes and to improve the overall quality and consistency of the codebase and of the documentation. This section is not intended to be exhaustive.

### General code comments

- The function `enforce_contiguos_epochs()` defined in `sc2sc/mod.rs` has a typo and should be renamed `enforce_contiguous_epochs()`
- A number of user messages in `demo-circuit/src/sc2sc/mod.rs` contain typos. For instance:
  - "Certificates should contains the same custom fields" on line 70
  - "Alloc current epoch `sc_tx_commitment_root` recostruction gadget" in `demo-circuit/src/sc2sc/mod.rs`, on lines 180 and 191
  - "Cannot retrive the forward transfer root" on lines 335-348
- In order to facilitate implementation and review of [ZenIP-42205](#), the naming of the variables between that document and the code should be consistent. For example, the variable `next_sc_tx_commitments_root_g` in the code is defined as `next_sc_tx_commitment_root` in ZenIP-42205 (note the missing `s` at `commitment`), which could wrongly indicate that the former consists of multiple commitments. This comment applies equally to the variable `next_sc_tx_commitment_root`.
- On line 171 of `src/sc2sc/mod.rs`, `"require(next_cert.previous_top_quality_hash == H(curr_cert_hash)" , H(curr_cert_hash) should be H(curr_cert) , as this represents the hash of the current certificate.`
- There is an inconsistency in the parameter order in functions `check_membership()` in the file `demo-circuit/src/sc2sc/mod.rs`. That function is implemented both for `ScCommitmentCertPathGadget`:

```
388 fn check_membership<CS: rics\_core::ConstraintSystemAbstract<FieldElement>>(<br>389     &self,<br>390     mut cs: CS,<br>391     sc_tx_commitment_root: &FpGadget<FieldElement>,<br>392     cert_hash: &FpGadget<FieldElement>,<br>393     sc_id: &FpGadget<FieldElement>,<br>394 )
```

and for `ScCommitmentCertPath`:

```
318 pub fn check_membership(<br>319     &self,<br>320     sc_tx_commitment_root: &FieldElement,<br>321     sc_id: &FieldElement,<br>322     cert_hash: &FieldElement,<br>323 )
```

Note that the order of the last two arguments (highlighted) is swapped between the two functions. This could lead to confusion, resulting in errors in using these functions.

- When calling the function `check_membership()` above, for example in the function `generate_constraints()` in the file `sc2sc/mod.rs`, the variable passed as parameter for the



Sidechain ID ( `sc_id` ) is the `ledger_id` , as can be seen in the highlighted portion of the code excerpt below.

```
.check_membership(  
    cs.ns(|| "Check current epoch sc_tx_commitment_root"),  
    &curr_sc_tx_commitments_root_g,  
    &curr_cert_hash_g,  
    &curr_cert_g.ledger_id_g,  
)?;
```

- The new function to instantiate a `Sc2Sc` circuit could use additional parameter validation steps. Currently, it only checks that the length of the `custom_fields` of the two certificates is valid, as can be seen in the code excerpt below.

```
67  assert_eq!(  
68      next_cert.custom_fields.len(),  
69      curr_cert.custom_fields.len(),  
70      "Certificates should contains the same custom fields"  
71  );  
72  assert!(next_cert.custom_fields.len() >= MIN_CUSTOM_FIELDS, "We need at least {}  
    ↳ custom fields: see  
73      https://github.com/HorizenOfficial/ZenIPs/blob/  
    ↳ 57fe28cb13202550ed29512f913de2508877dc0b/zenip-42205.md#zenip-42205  
74      for more details", MIN_CUSTOM_FIELDS);
```

- The link to ZenIP-42205 in the code excerpt above is outdated. The most current version is the following [zenip-42205.md](#); a number of changes were introduced between these two versions, specifically in the *Witnesses* and the *Statements* pertaining to the proof verification. The new content can be seen in bold text below.

Witnesses:

- **The ID of the SC ( `sc_id` )**
- A certificate for epoch N+1 ( `next_cert` ) (can be any certificate, not mandatory to be the top-quality one)
- Merkle path of `next_cert` data hash inside the subtree of certificates for the SC at epoch N+1 ( `next_cert_path` )
- **Merkle root of the subtree of forward transfers for the SC at epoch N+1 ( `next_fwt_root` )**
- **Merkle root of the subtree of backward transfers for the SC at epoch N+1 ( `next_bt_root` )**
- **Merkle path of the SC commitment inside the SC commitment tree at epoch N+1 ( `next_sc_comm_tree_path` )**
- Top quality certificate for epoch N ( `curr_cert` )
- Merkle path of `curr_cert` data hash inside the subtree of certificates for the SC at epoch N ( `curr_cert_path` )
- **Merkle root of the subtree of forward transfers for the SC at epoch N. ( `curr_fwt_root` )**
- **Merkle root of the subtree of backward transfers for the SC at epoch N. ( `curr_bt_root` )**
- **Sidechain creation transaction of the SC ( `scc_tx` )**
- **Merkle path of the SC commitment inside the SC commitment tree at epoch N+1 ( `curr_sc_comm_tree_path` )**



- Merkle path of the message to be redeemed inside `SC2SC_message_tree_root` present in the custom field of `curr_cert (msg_path)`

Statements:

- `next_cert_root = ApplyMerklePath(H(next_cert), next_cert_path)`
  - `curr_cert_root = ApplyMerklePath(H(curr_cert), curr_cert_path)`
  - `next_sc_root = H(next_fwt_root, next_bt_root, next_cert_root, scc_tx, sc_id)`
  - `curr_sc_root = H(curr_fwt_root, curr_bt_root, curr_cert_root, scc_tx, sc_id)`
  - `VrfyMembership(next_sc_root, next_sc_comm_tree_path, next_sc_tx_commitment_root) == TRUE`
  - `VrfyMembership(curr_sc_root, curr_sc_comm_tree_path, curr_sc_tx_commitment_root) == TRUE`
  - `next_cert.previous_top_quality_hash == H(curr_cert)`
  - `curr_cert.epoch == next_cert.epoch - 1`
  - `VrfyMembership(msg_hash, msg_path, curr_cert.SC2SC_message_tree_root) == TRUE`
- It appears that the computation of the Merkle tree roots (such as the ones computed for the message and for the commitment trees) do not utilize domain separators to differentiate between internal nodes and leaf nodes, which is a known avenue for second-preimage attacks<sup>3,4</sup>. This does not seem to be exploitable in the specific setting in which the SC2SC circuit uses them, since all the trees are of fixed size. However, if it were at all possible to dynamically set the tree height, or if future deployment decided to update that parameter, malicious participants may be able to exploit that oversight. Consider addressing that issue by domain-separating all hash function calls.

**Update:** The majority of issues highlighted here were fixed in [commit 0799b7a](#). Horizen Labs indicated that `ledger_id` and `sc_id` are synonymous variables. Additionally, they noted that the validation checks currently performed in `Sc2Sc::new()` already include all of the checks that an API user can perform, that cannot be enforced in the signature or within the circuit during proof creation. As such, no additional validation checks have been added to `Sc2Sc::new()`.

## Comments on ZenIP 42205

The document [ZenIP-42205](#) describes the proposed design of the cross-sidechain communication protocol. Some inconsistencies were identified, which are presented below.

- There is no *Terminology* section clearly defining the meaning of the keywords “MUST”, “MUST NOT”, “SHOULD”, and “MAY” as described in [RFC 2119](#) at the beginning of the document. Note that this section is defined in the [ZenIP template document](#).
- On the same topic, all the instances of “must” are not capitalized.
- Under *Entity definition*, the *cross chain message* paragraph describes the different attributes of a cross chain message. These attributes are untyped. In contrast, the description of the *redeem message* in the next paragraph defines types for its attributes, such as `curr_cert_hash: ByteArray`. However, in *redeem message*, the attribute `proof: ByteArray` is not accompanied with an explanation.
- The following statement is unclear:

3. <https://flawed.net.nz/2018/02/21/attacking-merkle-trees-with-a-second-preimage-attack/>

4. <https://bitslog.com/2018/06/09/leaf-node-weakness-in-bitcoin-merkle-tree-design/>



---

Every sidechain that wants to act as “sender” must emit certificates with the following structure:  $N + M = 32$  Where:  $M$  = number of bit vector elements  $N$  = number of certificate custom fields, all of them of fixed 32 bytes size.

More specifically, the meaning of  $N$  is well-understood, but the meaning of  $M$  isn't. Additionally, this statement does not seem to be enforced in the implementation.

- There is an extra whitespace in “cryptographic circuit , to allow”, which should be “cryptographic circuit, to allow”.
- In the sentence “The proof specification, for a message msg published in an epoch  $N$ ”, the variable  $N$  was already defined earlier to be the number of certificate custom fields.

## General Design Comments on the SC2SC Protocol

- Cross chain messages are composed of the following fields

◦ sendingScId,	◦ msgType,	◦ receiver,
◦ receivingScId,	◦ sender,	◦ payload.

Note that messages do not have any notion of creation or publication time. The ZenIP document states that

Leaves will be valorized in order of message publication inside the sidechain.

It is unclear how any ordering of the messages in a Merkle tree based on publication can be performed.

- Additionally, it seems that the field `msgType` in the message above could be part of the payload instead of being a component of the high-level message.
- The SC2SC Design document states

Since we build a merkle tree with the messages and we have a limited number of leaf on the tree, we need to limit the maximum number of messages created on every epoch.

For account model: The check is simpler: the limit can be checked directly inside the message processor. In case of limit overreached we may just do nothing and fail the tx execution.

It seems that this may allow attacker to perform denial of service attacks, by spamming the sidechain with messages. Consider ensuring that no such attacks may occur, and designing mitigations if this is a valid attack vector.

- The design document could include more specific details about what actions are and aren't allowed to be performed during the cross-chain message protocol, as well as what checks are in place to identify and protect against potentially malicious actions. For example, what if the sending and receiving sidechain IDs are the same in a cross-chain message? These types of threats as well as the mitigations in place should be documented somewhere, possibly in the design document.
- The definition of the redeem message in the Design Doc does not quite match that of ZenIP 42205:
  - Many of the fields are slightly differently named: `certificate_data_hash` vs `curr_cert_hash`, `next_certificate_hash` vs `next_cert_hash`, `sc_commitment_tree_root` vs `curr_sc_commitment_root`



- `next_certificate_hash` is erroneously specified as the hash of the next epoch *top* quality certificate in the Design Document. The zenip 42205 only requires `next_cert_hash` to be the hash of *any* certificate in the next epoch. This corresponds with the figure on page 11 of the Design Doc.

## Notes on FFI Rust code

- PR 110 exposes several Rust methods to Java via FFI. For example, it is possible to verify commitment certificate paths, (de)serialize and drop them from Rust. It is necessary to expose methods that free memory in Rust for objects that are not dropped by Rust automatically. In `api/src/lib.rs`, such functions are added consistently for each new introduced struct (see also a more general paper<sup>5</sup> on memory safety in the context of FFI). It is up to the Java code to correctly call these methods to avoid memory leaks.

An additional complication when evaluating the code for leaks is that apart from FFI, `api/src/lib.rs` uses JNI in order to call back into JVM. For example, the `nativeSerializePublicKey` FFI function in `lib.rs` in turn calls a Java function over JNI (inside `byte_array_from_slice`):

```
/// Create a new java byte array from a rust byte slice.
pub fn byte_array_from_slice(&self, buf: &[u8]) -> Result<jbyteArray> {
    let length = buf.len() as i32;
    let bytes: jbyteArray = self.new_byte_array(length)?;
    jni_unchecked!(
        self.internal,
        SetByteArrayRegion,
        bytes,
        0,
        length,
        buf.as_ptr() as *const i8
    );
    Ok(bytes)
}
```

The idea is to create a new byte array using JNI's `NewByteArray` in JVM and then set it up using `SetByteArrayRegion`. There do *exist* JNI functions for releasing memory, e.g., `ReleaseByteArrayElements`. However, in this case it appears that as the address of the created byte array will be created inside and returned to the original JVM, it is not necessary to call the release functions.

- It is worth noting that the following function can be used from Java to validate that two field scalars are equal:

```
274 ffi_export!(
275     fn Java_com_horizen_librustsidechains_FieldElement_nativeEquals(
276         _env: JNIEnv,
277         // this is the class that owns our
278         // static method. Not going to be
279         // used, but still needs to have
280         // an argument slot
281         _field_element_1: jobject,
```

---

5. *Detecting Cross-Language Memory Management Issues in Rust*, Zhuohua Li, Jincheng Wang, Mingshen Sun and John C.S. Lui, <https://zhuohua.me/assets/ESORICS2022-FFIChecker.pdf>



```
282     _field_element_2: JObject,  
283 ) -> jboolean {  
284     // ...SNIP...  
285     match field_1 == field_2 {
```

The function performs a non-constant time equality check. Field scalars may be used to hold secret material. The non-constant time comparison takes place as follows:

```
impl Ord for $name {  
    #[inline]  
    fn cmp(&self, other: &Self) -> ::std::cmp::Ordering {  
        for (a, b) in self.0.iter().rev().zip(other.0.iter().rev()) {  
            if a < b {  
                return ::std::cmp::Ordering::Less;  
            } else if a > b {  
                return ::std::cmp::Ordering::Greater;  
            }  
        }  
  
        ::std::cmp::Ordering::Equal  
    }  
}
```

The leak appears minor, as it is cascaded over 64-bit values. However, it is worth noting that the iterator is reversed and the number is stored in little-endian representation. Depending on the modulus, the u64 that's verified first may be significantly less than 64 bits, making the comparison more amenable to timing leaks on the most significant limb.

**Update:** Horizen Labs indicated that they are aware of the highlighted memory considerations when using JVM FFI code in Rust, and that they have taken steps to mitigate potential issues by using a careful internal review process. Additionally, they noted that they are aware of the non-constant time equality check in the `Java_com_horizen_librustsidechains_FieldElement_nativeEquals` function, and that this function is only used for test purposes.



## 7 Contact Info

---

The team from NCC Group has the following primary members:

- Elena Bakos Lang – Consultant  
[elena.bakoslang@nccgroup.com](mailto:elena.bakoslang@nccgroup.com)
- Paul Bottinelli – Consultant  
[paul.bottinelli@nccgroup.com](mailto:paul.bottinelli@nccgroup.com)
- Aleksandar Kircanski – Consultant  
[aleksandar.kircanski@nccgroup.com](mailto:aleksandar.kircanski@nccgroup.com)
- Javed Samuel – Cryptography Services Practice Director  
[javed.samuel@nccgroup.com](mailto:javed.samuel@nccgroup.com)

The team from Zen Blockchain Foundation has the following primary members:

- Giacomo Gussoni  
[giacomo@horizenlabs.io](mailto:giacomo@horizenlabs.io)
- Daniele Di Benedetto  
[daniele@horizenlabs.io](mailto:daniele@horizenlabs.io)

