



Horizen EVM Sidechain Cryptolib Cryptography and Implementation Review

Zen Blockchain Foundation
Version 1.0 – May 10, 2023

©2023 – NCC Group

Prepared by NCC Group Security Services, Inc. for Horizen Labs. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Elena Bakos Lang
Paul Bottinelli
Eric Schorn

Prepared For
Giacomo Gussoni
Daniele Di Benedetto

1 Table of Contents

1	Table of Contents	2
2	Executive Summary	3
2.1	Synopsis	3
2.2	Scope	3
2.3	Key Findings	3
2.4	Strategic Recommendations	4
3	Dashboard	5
4	Table of Findings	6
5	Finding Details	7
6	Finding Field Definitions	50
6.1	Risk Scale	50
6.2	Category	51
7	Non-Security Impacting Observations	52
7.1	Methodology Notes	52
7.2	On Threshold Signature Verification	52
7.3	Inconsistent Documentation for NULL Signatures	54
7.4	Confusing Endianness in Field Element Representations	55
7.5	On Merkle Tree Primitives	57
7.6	R1CS Gadgets Implementation Notes	57
7.7	Poseidon Parameters	58
7.8	Bit-length-based Range Checks	59
7.9	Elliptic Curve Point Representation and Coordinate Systems	60
7.10	Bowe-Hopwood Hash	60
7.11	Schnorr Signatures	61
7.12	Constraint System Primitives	61
8	Circuit Design	63
9	Appendix: Additional Scoping Detail	67



2 Executive Summary

Synopsis

Starting in December 2022, Horizen Labs engaged NCC Group's Cryptography Services team to perform a security review of several portions of the *ginger-lib* and *zendoo-sc-cryptolib* source code repositories. The *ginger-lib* repository contains a Rust library providing the low-level primitives and gadgets needed to build recursive succinct zero-knowledge arguments, while the *zendoo-sc-cryptolib* repository contains a Rust crate that exposes components of *ginger-lib* that are needed by the Zendoo Sidechains SDK, through the Java Native Interface (JNI), as well as sample arithmetic circuits built on top of these. These codebases provide critical functionalities that are vital to the basic operation and trustworthiness of the Horizen blockchain and sidechains.

This engagement comprised an initial testing part and a retesting part. Initial testing was split into two phases: the first phase was two weeks long and was delivered in December 2022, while the second phase was four weeks long and was delivered in February 2023.

Following this initial testing, in early March 2023, the NCC Group team performed a retest of the findings uncovered during the first two phases. A total of 55 person-days of effort were expended on the project.

Scope

The code for *ginger-lib* is located on GitHub at <https://github.com/HorizenOfficial/ginger-lib/>, with the review performed on branch `development` at commit ID `c0f35a95` for the first phase, and at commit ID `25384653` for the second phase.

The code for *zendoo-sc-cryptolib* is located on GitHub at <https://github.com/HorizenOfficial/zendoo-sc-cryptolib/>, with the review performed on branch `development` at commit ID `d28ad599` for the first phase, and at commit ID `20074639` for the second phase.

The scope includes the following elements:

- Relevant portions of *ginger-lib* (including the Schnorr, Verifiable Random Function (VRF), and Merkle tree implementations and their supporting types), further detailed under [Appendix: Additional Scoping Detail](#);
- Relevant portions of *zendoo-sc-cryptolib* (including the *demo-circuit* folder, and specifically the `naive_threshold_sig_w_key_rotation` circuit and all its supporting types, traits and functions, and the *jni* and *api* folders), further detailed under [Appendix: Additional Scoping Detail](#).

One additional resource was shared to support the review: an internal document describing the design of the naive threshold with key rotation circuit and entitled *circuit_design.txt*, which was copied in the appendix [Circuit Design](#).

Due to the size and complexity of the different code bases under review, the NCC Group team focused their efforts on the scope described above and did not venture outside of the specific repositories listed. Overall, good coverage was achieved on the items in scope.

Additionally, issues may arise in the deployment of circuits and other systems based on the libraries that were reviewed. These issues are inherently outside of the scope of the current review.

Key Findings

The initial testing phases of the assessment uncovered a number of application flaws. The most notable findings were:

- **VRF Violates Uniqueness Assurance:** Several significant departures from the well-known and broadly reviewed draft IRTF CFRG VRF specification invalidate the



underlying security proofs and weaken the required assurances provided by the VRF functionality.

- **Non-Injective `encode_to_curve` Violates VRF Collision Resistance Assurance:** A non-injective `encode_to_curve` function causes the VRF to violate the collision resistance assurance required to avoid a proof corresponding to multiple values.
- Some Merkle tree-related findings, including **Inadequate Validator Keys Merkle Tree** where the design and implementation of the validator keys Merkle tree may be vulnerable to replay attacks, and **Missing Domain Separators in Merkle Trees** where an attacker may be able to produce a series of leaves which allows them to forge an inclusion proof in the Merkle tree.
- Some field arithmetic and elliptic curve operation findings, including **Potentially Problematic Instantiations of Field Elements from Negative Values** where the instantiation of field elements from negative values may result in inconsistent arithmetic computation outcomes, **Incomplete Elliptic Curve Formulas** where the lack of complete addition formulas can lead to unexpected issues, and **Problematic Point at Infinity Representation and Comparison** where arbitrary representations of the point at infinity, as well as missing checks for this special case in the equality function, may introduce subtle vulnerabilities in cryptographic operations.

Some additional, informational review notes are provided in the appendix [Non-Security Impacting Observations](#).

After retesting, NCC Group found that the majority of the findings had been addressed. Out of a total of seventeen (17) original findings, twelve (12) were marked as *Fixed*, four (4) were marked as *Risk Accepted* (with reasonable explanations from Horizen Labs) and one (1) informational finding was marked as *Partially Fixed*. Additionally, the NCC Group team noted that Horizen Labs diligently addressed many of the observations presented in the informational Engagement Notes section.

Strategic Recommendations

Generally, documentation was found to be scarce, particularly for some of the more complex mathematical and cryptographic operations, or the constraints in the proof system. As the code base under review is a set of libraries, clear documentation is very important in order for the end user of these functionalities to avoid introducing vulnerabilities in the systems they build on top. Consider writing documentation for the different high-level functions, as well as more detailed comments in the source code itself when complex operations are performed.

Additionally, the code base implements a few state-of-the-art cryptographic algorithms that may not have received the level of scrutiny that other, more established primitives have. Consider being even more diligent in the documentation efforts regarding these algorithms, and clearly keep track of the assumptions and modifications that have been introduced.

The libraries in scope use at least four different and non-compatible endianness representations, as also discussed in the appendix [Non-Security Impacting Observations](#). This seems to contribute significantly to the complexity and the verbosity of the code, since a lot of operations have to be preceded by conversions. A worthy (although non-trivial) development effort could be to tackle this issue, and at least ensure that the representation of elements within a given library is consistent.

Remove all dead code as well as currently unused functionalities, as they only increase attack surface. Consider implementing test coverage analysis and writing a significant amount of additional unit test cases in order to increase coverage.



3 Dashboard

Target Data

Name	Zendoo Sidechain Cryptolib
Type	Cryptographic Libraries
Platforms	Rust, Java
Environment	Local Instance


















Engagement Data

Type	Cryptography Implementation Review
Method	Source Code Security Review
Dates	2022-12-05 to 2023-02-17
Consultants	3
Level of Effort	55 person-days

















Targets

zendoo-sc-cryptolib	A Rust crate that exposes the <i>ginger-lib</i> components needed by the Zendoo sidechain SDK to Java, through JNI: https://github.com/HorizenOfficial/zendoo-sc-cryptolib
ginger-lib	A general purpose zk-SNARK library supporting recursive proof composition: https://github.com/HorizenOfficial/ginger-lib


















Finding Breakdown

Critical issues	0	
High issues	1	
Medium issues	7	      
Low issues	4	   
Informational issues	5	    
Total issues	17	

Category Breakdown

Cryptography	12	          
Data Exposure	1	
Data Validation	2	 
Error Reporting	1	
Patching	1	

Component Breakdown

Systemic	2	 
ginger-lib	10	         
zendoo-sc-cryptolib	5	    

 Critical  High  Medium  Low  Informational



4 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
VRF Violates Uniqueness Assurance	Fixed	LK6	High
Inadequate Validator Keys Merkle Tree	Fixed	NLU	Medium
Potentially Problematic Instantiations of Field Elements from Negative Values	Fixed	YX2	Medium
Missing Domain Separators in Merkle Trees	Risk Accepted	9D7	Medium
Missing Enforcement of Minimum Seed Size in KeyGen	Risk Accepted	XX3	Medium
Problematic Point at Infinity Representation and Comparison	Fixed	HWH	Medium
Non-Injective <code>encode_to_curve</code> Violates VRF Collision Resistance Assurance	Fixed	HHX	Medium
Incomplete Elliptic Curve Formulas	Fixed	39X	Medium
Insecure Public Keys or Signatures can be Generated for Schnorr Signatures	Fixed	XKT	Low
Silent <code>pow()</code> Overflow in Merkle <code>width</code> and <code>height</code>	Fixed	9HX	Low
Confusing Endianness in <code>UInt8</code> Vector Creation	Fixed	LUC	Low
Panic in Field Element Gadget Deserialization from Empty Array	Fixed	74E	Low
Hashing to Elliptic Curve is Biased and Not Standard	Risk Accepted	6AH	Info
Outdated Rust Toolchain and Dependencies	Partially Fixed	9XW	Info
VRF Implementation Anomalies	Fixed	PW9	Info
Weak Error Handling via <code>unwrap()</code>	Fixed	TR4	Info
Missing Memory Zeroization on Secret Keys	Risk Accepted	YWJ	Info



5 Finding Details

High

VRF Violates Uniqueness Assurance

Overall Risk High
Impact High
Exploitability High

Finding ID NCC-E005939-LK6
Component ginger-lib
Category Cryptography
Status Fixed

Impact

Several significant departures from the well-known and broadly reviewed draft IRTF CFRG VRF specification may invalidate the underlying security proofs and weaken the assurances provided by the VRF functionality. For example, multiple different intermediate $\{\gamma, c, s\}$ proofs can be generated that will successfully verify and will lead to different final results under the same message and public key. This violates the VRF assurance of uniqueness and allows the VRF to be re-run until favorable 'random' results are obtained.

Description

The Horizen VRF functionality is intended to follow the general flow of the well-known and broadly reviewed draft IRTF CFRG VRF specification¹ but not perfectly match it in every respect due to a different context and constraints. However, the code contains several significant points of departure: 1) γ (Γ in the specification) is missing from the challenge generation, 2) the nonce r is randomly generated rather than deterministic, 3) the hash is missing domain separators, and 4) the code's 'primary API' is structured differently.

A portion of the logic from the `prove()` function implemented in `primitives/src/vrf/ecvrf/mod.rs`² is excerpted below.

```
258 //Compute mh = hash_to_curve(message)
259 let message_on_curve =
260     GH::evaluate(group_hash_params, to_bytes!(&message).unwrap().as_slice())?;
261
262 //Compute gamma = message_on_curve^sk
263 let gamma = message_on_curve.mul(sk);
264
265 let required_leading_zeros_c = compute_truncation_size(
266     F::size_in_bits() as i32,
267     G::ScalarField::size_in_bits() as i32,
268 );
269
270 let required_leading_zeros_s = compute_truncation_size(
271     G::ScalarField::size_in_bits() as i32,
272     F::size_in_bits() as i32,
273 );
274
275 let (c, s) = loop {
276     //Choose random scalar
277     let r = G::ScalarField::rand(rng);
278
279     //Compute a = g^r
```

1. <https://datatracker.ietf.org/doc/draft-irtf-cfrg-vrf/>

2. <https://github.com/HorizenOfficial/ginger-lib/blob/25384653cc9818bbecea1108ef31e5f3930758cf/primitives/src/vrf/ecvrf/mod.rs#L258-L294>



```

280     let a = G::prime_subgroup_generator().mul(&r);
281
282     //Compute b = message_on_curve^r
283     let b = message_on_curve.mul(&r);
284
285     //Compute c = H(m || pk.x || a.x || b.x)
286     let c = {
287         let mut digest = FH::init_constant_length(4, None);
288         digest
289             .update(message)
290             .update(pk.0.to_field_elements()?.[0])
291             .update(a.to_field_elements()?.[0])
292             .update(b.to_field_elements()?.[0])
293             .finalize()

```

First, the highlighted comment on line 285 above summarizes the implemented `challenge_generation()` functionality in the subsequent code statement, where `gamma` is **not included** in the hash function. The corresponding procedural step in the VRF specification is shown below that **does include** `gamma`.

`c = ECVRF_challenge_generation(Y, H, Gamma, k*B, k*H)` (see Section 5.4.3)

Second, the specification describes a deterministic nonce generation similar to that in RFC 6979³. However, the code shows `r` as being randomly chosen as highlighted on line 277 above. This means the `prove()` function can be rerun several times, with different `r` values chosen to generate different proof outputs `s` and `c`, making the proof malleable. A malicious prover could choose different `r` for each of the point multiplications on lines 280 and 283 if desired.

Given the missing `gamma` and the flexibility of choosing `r` values, consider proofs constructed with the following form (where `cInv` is `c-1`, `m` is the message hashed to a point, `g` is a generator, and `sk/pk` are the secret and public keys):

$$\{\text{gamma} = m^{(\text{sk} + \text{cInv})}, c = H(m \parallel pk \parallel g^{(r+1)} \parallel m^r), s = r + (\text{sk} + \text{cInv}) * c\}$$

Lines 340 and 343 of the `proof_to_hash()` function⁴ implementation excerpted below (which includes a gating verify step) calculate the two following critical values used for verification:

$$\begin{aligned} u &= g^s - pk^c = g^{(r + (\text{sk} + \text{cInv}) * c)} - g^{\text{sk} * c} = g^{(r + \text{sk} * c + 1 - \text{sk} * c)} = g^{(r+1)} \\ v &= m^s - \text{gamma}^c = m^{(r + (\text{sk} + \text{cInv}) * c)} - m^{(\text{sk} + \text{cInv}) * c} = m^{(r + \text{sk} * c + 1 - \text{sk} * c - 1)} = m^r \end{aligned}$$

The above values match the latter two values used in the initial hash construction, so the `c == c'` test will pass below on line 358. However, multiple runs with different `r` will result in different `gamma` values for the same message and public key. These are then hashed into different final results which violates the VRF assurance of uniqueness.

3. <https://www.rfc-editor.org/rfc/rfc6979>

4. <https://github.com/HorizenOfficial/ginger-lib/blob/25384653cc9818bbecea1108ef31e5f3930758cf/primitives/src/vrf/ecvrf/mod.rs#L330-L376>



Third, the specification of the `ECVRF_challenge_generation()`⁵ function contains domain separators which are missing from the code. Without domain separators, calls to different random oracles may result in the same output, thus contradicting the random oracle model on which the security proofs are built. This element is a lower-risk aspect of this finding.

Fourth, the specification articulates distinct `ECVRF_proof_to_hash()` and `ECVRF_verify()` functions, while the code (as noted above) combines these into a single `proof_to_hash()` function. While this refactoring itself has minimal impact, the structure may introduce interpretation weaknesses such as mistaking multiple `c` and `s` values (with the same `gamma`) as a violation of uniqueness. Note that VRF uniqueness is only stipulated to hold for the final hash. This element is an informational aspect of this finding.

```
330 //Compute mh = hash_to_curve(message)
331 let message_on_curve =
332     GH::evaluate(group_hash_params, to_bytes!(&message).unwrap().as_slice());
333
334 let c_bits = proof.c.write_bits();
335 let s_bits = proof.s.write_bits();
336 let c_conv = convert::<G::ScalarField>(c_bits)?;
337 let s_conv = convert::<G::ScalarField>(s_bits)?;
338
339 //Compute u = g^s - pk^c
340 let u = G::prime_subgroup_generator().mul(&s_conv) - &(pk.0.mul(&c_conv));
341
342 //Compute v = mh^s - gamma^c
343 let v = message_on_curve.mul(&s_conv) - &proof.gamma.mul(&c_conv);
344
345 //Compute c' = H(m || pk.x || u.x || v.x)
346 let c_prime = {
347     let mut digest = FH::init_constant_length(4, None);
348
349     digest
350         .update(message.clone())
351         .update(pk.0.to_field_elements()?[0])
352         .update(u.to_field_elements()?[0])
353         .update(v.to_field_elements()?[0])
354         .finalize()
355 }?;
356
357 //Verify valid proof
358 match proof.c == c_prime {
359 ...
```

The overall impact of violating the uniqueness assurance with the multiple `{gamma, c, s}` construction shown above is severe because a malicious prover can rerun the randomness calculations until a favorable result is obtained.

Recommendation

Follow the IRTC CFRG VRF specification⁶ more closely. Specifically, incorporate `gamma` into challenge generation, utilize a deterministically generated `r`, and include domain separators.

5. <https://www.ietf.org/archive/id/draft-irtf-cfrg-vrf-15.html#name-ecvrf-proving> and <https://www.ietf.org/archive/id/draft-irtf-cfrg-vrf-15.html#name-ecvrf-challenge-generation>

6. <https://datatracker.ietf.org/doc/draft-irtf-cfrg-vrf/>



Location

ginger-lib/primitives/src/vrf/ecvrf/mod.rs

Retest Results

2023-03-01 – Fixed

The Horizen Labs team introduced changes in two commits ([commit 3d3427e7](#) for [ginger-lib](#) and [commit ddc1657](#) for [zendoo-sc-cryptolib](#)) that address the issues outlined in this finding.

The implementation of the VRF is now much more aligned with the VRF draft specification. Specifically, the challenge computation now incorporates `gamma`, as can be seen in the code excerpt below.

```
let hash_input = vec![FH::Data::from(CHALLENGE_DOMAIN_SEPARATOR as u128), message].into_iter()
    .chain(pk.0.to_field_elements())
    .chain(gamma.to_field_elements())
    .chain(a.to_field_elements()?.into_iter())
    .chain(b.to_field_elements()?.into_iter())
    .collect::<Vec<_>>();
```

Domain separators have also been added to the computation of the challenge and in the final proof-to-hash generation (an example of which can also be seen in the code snippet above).

Additionally, the generation process for the nonce `r`, which was previously randomly sampled (as outlined on line 277 highlighted in the first code snippet in the description of this finding above) has now been replaced by a deterministic process, following [RFC 6979 \(Section 3.2\)](#), as proposed in the VRF specification. This process is carried out by the `generate_nonce()` function defined in the file `primitives/src/vrf/ecvrf/nonce_generation.rs`. This file also introduces an HMAC implementation, which is used by RFC 6979. This HMAC primitive was taken from the RustCrypto crate [rfc6979](#).

These changes are in line with the recommendation above, and, barring a few presumably minor differences with the VRF specification (such as the missing trailing domain separators both in the challenge computation and the output hash, and a small difference in the ordering of the fields upon challenge generation), the code is now better aligned with that specification.

As a result, this finding is considered *fixed*.

The new primitives have been introduced without accompanying tests; consider adding unit tests ensuring the correct behavior of the new code.



Inadequate Validator Keys Merkle Tree

Overall Risk Medium
Impact Medium
Exploitability Medium

Finding ID NCC-E005939-NLU
Component zendoo-sc-cryptolib
Category Cryptography
Status Fixed

Impact

The current design and implementation of the validator keys Merkle tree may be vulnerable to forced key rotation and replay attacks. If successfully exploited, an attacker could disrupt the key rotation procedure, deny service to other parties, and eventually destabilize trustworthiness in the system for all relying parties.

Description

In the circuit design of the sidechain cryptolib, validators possess two keys: a signing key and a master key. In short, the former allows validators to sign withdrawal certificates, while the latter is necessary for key rotation purposes. Both keys are used to authenticate updates to the key material. To support the rotation of these keys, withdrawal certificates contain the root of a Merkle tree containing the keys of all validators for the next epoch; that root is then verified during SNARK verification to ensure potential key rotations have been performed legitimately.

Cryptographic key management is a complex topic and the NCC Group team noted some potential shortcomings at the design level with the current validator keys Merkle tree construction.

Consider the `get_validators_key_root()` function, excerpted from the file [naive_threshold_sig_w_key_rotation/data_structures.rs](#), and provided below for reference. This function iterates over all keys (starting on line 156 below), hashes them, and appends them (see highlighted lines 159-163) to a list of leaves which then recursively gets hashed in order to obtain the Merkle root, via the call to the function `finalize_in_place()` (on line 172).

```

137 pub(crate) fn get_validators_key_root(
138     max_pks: usize,
139     sig_keys: &[FieldBasedSchnorrPk<G2Projective>],
140     master_keys: &[FieldBasedSchnorrPk<G2Projective>],
141 ) -> Result<FieldElement, Error> {
142     let height = ((max_pks.next_power_of_two() * 2) as f64).log2() as usize;
143     let null_leaf: FieldElement = GingerMHTParams::ZERO_NODE_CST.unwrap().nodes[0];
144     let mut tree = GingerMHT::init(height, 1 << height)?;
145
146     let get_key_hash = |pk: &FieldBasedSchnorrPk<G2Projective>| ->
147         ↳ Result<FieldElement, Error> {
148         let pk_fe = pk.to_field_elements()?;
149
150         let mut h = FieldHash::init_constant_length(pk_fe.len(), None);
151         pk_fe.into_iter().for_each(|fe| {
152             h.update(fe);
153         });
154         h.finalize()
155     };
156     for i in 0..max_pks.next_power_of_two() {

```



```

157         if i < sig_keys.len() {
158             // Compute curr pks hash and append them to curr tree
159             let signing_key_hash = get_key_hash(&sig_keys[i])?;
160             let master_key_hash = get_key_hash(&master_keys[i])?;
161
162             tree.append(signing_key_hash)?;
163             tree.append(master_key_hash)?;
164         } else {
165             // Pad trees with null leaves if max_pks is not a power of two
166             tree.append(null_leaf)?;
167             tree.append(null_leaf)?;
168         }
169     }
170
171     // Finalize the trees and get the root
172     tree.finalize_in_place()?;
173     tree.root()
174     .ok_or_else(|| Box::<dyn std::error::Error>::from("cannot get merkle root"))
175 }

```

First, note that there is no distinction when hashing a signing key and a master key, such as a domain separator. This observation applies both to hashed keys used in the computation of the root of the Merkle tree and the hashed keys being signed for the key rotation process. As such, if a validator were to use a single key for both uses, the hashed value in the tree would be the same. While this is not necessarily problematic in itself, if the validators were then to rotate one of their keys, an attacker could tamper with that request to rotate the other key (since the hashes being signed are not differentiated) without it being detected by the proof verification.

After reporting this issue, the Horizen Labs team also highlighted another concrete attack arising from the lack of domain separation. Namely, an attacker observing the rotation of one key can unconditionally trigger a key update of the other key to the same key.

Additionally, the Merkle root computation is not tied to any epoch or time period. As such, state transitions could be replayed; a single validator rotating keys back to previously used keys could then have an earlier key rotation message be replayed, leading to an unwanted key update.

In case the number of validators is not an exact power of two, the rest of the leaves are populated with pre-defined, empty values, as can be seen on lines 166 and 167 above, where the value `null_leaf` is appended to the list of leaves. The `null_leaf` variable is assigned the value `ZERO_NODE_CST` from the *ginger-lib* dependency. The documentation for this constant value, defined in *ginger-lib/primitives/src/merkle_tree/field_based_mht/mod.rs* and shown below, states that it corresponds to the “pre-computed hash of the empty node”:

```

/// Definition of parameters needed to implement and optimize a Merkle Tree whose nodes and
↳ leaves
/// are Field elements. The trait is generic with respect to the arity of the Merkle Tree.
pub trait FieldBasedMerkleTreeParameters: 'static + Clone {
    type Data: Field;
    /// Actually unnecessary, but simplifies the overall design
    type H: FieldBasedHash<Data = Self::Data>;
    /// The arity of the Merkle Tree

```



```
const MERKLE_ARITY: usize;
/// The pre-computed hashes of the empty nodes for the different levels of the Merkle Tree
const ZERO_NODE_CST: Option<FieldBasedMerkleTreePrecomputedZeroConstants<'static,
↳ Self::H>>;
}
```

However, this value is actually the zero field element, and not the hash thereof.

Finally, it seems that the Merkle tree root computed by the `finalize_in_place()` call does not distinguish leaves from internal nodes, which is a known avenue for second-preimage attacks^{7,8}.

Recommendation

- Add a domain separation tag to distinguish when hashing signing or master keys.
- Consider explicitly tying the Merkle tree root to an epoch, in order to avoid replay attacks, for example by introducing the epoch number in the root computation.
- Consider adding domain separation tags to distinguish hashing of nodes and leaves, although executing such an attack in this case seems slightly contrived.
- Update the documentation to indicate that it is not the *hash* of the empty value that is stored, but the zero field element.

Reproduction Steps

Adding the debug statement highlighted below results in the following value being printed out, `Fp256(BigInteger256([0, 0, 0, 0]))`, showcasing that this value is indeed 0, and not the hash of 0.

```
pub(crate) fn get_validators_key_root(
    max_pks: usize,
    sig_keys: &[FieldBasedSchnorrPk<G2Projective>],
    master_keys: &[FieldBasedSchnorrPk<G2Projective>],
) -> Result<FieldElement, Error> {
    let height = ((max_pks.next_power_of_two() * 2) as f64).log2() as usize;
    let null_leaf: FieldElement = GingerMHTParams::ZERO_NODE_CST.unwrap().nodes[0];
    let mut tree = GingerMHT::init(height, 1 << height)?;
    println!("{}", null_leaf);
}
```

Location

zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/
data_structures.rs

Retest Results

2023-03-03 – Fixed

In response to this finding, the Horizen Labs team updated the key rotation protocol design. The key update signatures are no longer computed solely on the fresh keys, but additional context is now included in the payload to be signed. Specifically, the message to be signed is no longer `H(new_key)`, but is computed as:

```
H(H(new_key) || key_type || salt || epoch_id || ledger_id)
```

where

- `key_type` is a domain separation byte set to `"s"` for signing key updates, and to `"m"` for master key updates;

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

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



-
- `salt` acts as a version number, currently set to `0`, which binds each key update payload to a specific version of the protocol, preventing replay attacks across different protocol versions;
 - `epoch_id` binds the signature to the current epoch, preventing replay attacks;
 - `ledger_id` binds the signature to a specific sidechain.

While the Merkle tree design was not modified, the additional context added to the signatures seems to prevent the attacks described in this finding and are aligned with the recommendations provided above. Specifically, the domain separation tags now prevent attackers swapping updates between master and signing keys, and the different `ids` prevent replay of key update messages. Provisions for future protocol updates have also been included by means of the `salt` value.

[Pull request 102](#) implements these changes. Since it would be much more difficult for attackers to force key rotation through exploitation of this issue, this finding is considered *fixed* as a result.



Potentially Problematic Instantiations of Field Elements from Negative Values

Overall Risk Medium

Impact Medium

Exploitability Low

Finding ID NCC-E005939-YX2

Component zendoo-sc-cryptolib

Category Cryptography

Status Fixed

Impact

The instantiation of field elements from negative values may result in inconsistent arithmetic computation outcomes, which could in turn lead to numerous adverse and unforeseen consequences, such as threshold signature verification bypass.

Description

Arithmetic operations using instantiation of `FieldElements` from unsigned `long s` may lead to potentially diverging results, depending on how said elements were instantiated. Field elements are generated by the macro `impl_Fp!` (in `algebra/src/fields/models/mod.rs` of *ginger-lib*) and are represented by custom `BigInteger` types under the hood (generated by the macro `bigint_impl!` in the file `algebra/src/biginteger/macros.rs` of *ginger-lib*). These `BigInteger s` essentially wrap arrays of `u64` elements. As an example, the representation of 0 as an `Fp256` field element is given by `Fp256(BigInteger256([0, 0, 0, 0]))`.

Field elements can be instantiated from unsigned integers, via the constructor function `FieldElement::from(value as u64)`, which converts the `value` passed as parameter to a `BigInteger`, and uses that result to instantiate an element of the correct field. However, the wrap-around behavior of unsigned integer arithmetic operations can lead to adverse and unforeseen consequences.

As an example, the field element constructed as `f1 = FieldElement::from(1) - FieldElement::from(2)` will differ from the field element `f2 = FieldElement::from(1 - 2)`. Specifically, the first field element (`f1`) will be equal to the field order minus one. In contrast, the second field element will be constructed from the value `-1`, which, due to Rust's wrap-around arithmetic, will be converted to the quantity `u64::MAX - 1` first, before being used to instantiate a `BigInteger`. Hence, the following equation *does not* hold:

```
FieldElement::from(1) - FieldElement::from(2) == FieldElement::from(1 - 2)
```

The code base makes heavy use of the `FieldElement::from()` pattern, and in some cases seems to rely on the assumption that the above equation holds. Specifically, the `new()` function used to instantiate a `NaiveThresholdSignatureWKeyRotation` in the `demo-circuit/src/naive_threshold_sig_w_key_rotation/mod.rs`, uses the following code excerpt.

```
let threshold = FieldElement::from(threshold);

// <snip>

//Compute b as v-t and convert it to field element
let b_bits = (FieldElement::from(valid_signatures as u64) - threshold).write_bits();
let to_skip = FieldElement::size_in_bits() - (log_max_pks + 1);
```



In case the number of valid signatures is smaller than the threshold, this corresponds to the left-hand side of the equation above, where two field elements are instantiated and then subtracted.

In contrast, in the file `api/src/cctp_calls.rs`, which corresponds to the right-hand side of the example equation above, the function `get_naive_threshold_sig_circuit_prover_data()` computes the subtraction of the two `u64` values before instantiating a field element with that value, see below.

```
//Compute b as v-t and convert it to field element
let b = FieldElement::from(valid_signatures - threshold);
```

Interestingly, this behavior is only visible in release mode, provided that `valid_signatures - threshold` is less than 0. Indeed, Rust performs wrapping arithmetic in release mode, while it crashes with a panicked at 'attempt to subtract with overflow' in debug mode.

Additionally, discrepancies and unexpected adverse effects may also result in instantiation of `FieldElement` objects from the corresponding Java class (in `jni/src/main/java/com/horizen/librustsidechains/FieldElement.java`), specifically when creating objects from the `createFromLong(long value)` constructor, which accepts signed `long` values but silently converts negative arguments to their respective unsigned representations.

Recommendation

The essence of this finding is that unsigned integer arithmetic wraps automatically in release mode, thereby silently succeeding when potentially incorrect computations should arguably be flagged. As such, consider inspecting all instances of calls to

`FieldElement::from()` to check whether arithmetic operations involving the argument of that function have been performed, and whether the result could have wrapped around.

Additionally, consider replacing arithmetic operations with their wrapping counterpart (such as `wrapping_sub()`⁹) and write extensive tests to ensure that no unexpected discrepancies between debug and release mode goes unnoticed.

Finally, carefully evaluate whether instantiations of Java `FieldElement` objects should be able to be created using negative values, and consider adding checks to the constructor to prevent unforeseen usages.

Location

- [zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/mod.rs:L169](#)
- [zendoo-sc-cryptolib/api/src/cctp_calls.rs:L220](#)
- [zendoo-sc-cryptolib/jni/src/main/java/com/horizen/librustsidechains/FieldElement.java](#)

Retest Results

2023-03-06 – Fixed

In [pull request 100](#), the Horizen Labs team fixed the specific problematic instantiation highlighted above (in the file `api/src/cctp_calls.rs`) by first converting the `threshold` to a field element, as follows.

```
let b = FieldElement::from(valid_signatures) - FieldElement::from(threshold);
```

The team also added documentation to the Java `FieldElement.createFromLong()` function specifying that the input parameter should be a positive integer.

9. https://doc.rust-lang.org/std/primitive.u64.html#method.wrapping_sub



Although additional tests *could* be written to help detect and limit the misuses of that pattern, the problematic instance identified in this finding was addressed. This finding is considered *fixed* as a result.



Missing Domain Separators in Merkle Trees

Overall Risk Medium

Impact High

Exploitability Low

Finding ID NCC-E005939-9D7

Component ginger-lib

Category Cryptography

Status Risk Accepted

Impact

An attacker may be able to produce a series of leaves which allows them to forge an inclusion proof in the Merkle tree.

Description

The Merkle tree logic implemented in *ginger-lib/primitives/src/merkle_tree/field_based_mht/append_only/mod.rs*, *naive/mod.rs* and *big_lazy_merkle_tree.rs* does not utilize domain separators to differentiate between internal nodes and leaf nodes. Code comments from the former source file are excerpted below, so this is a known issue¹⁰.

```
/// WARNING. This Merkle Tree implementation:  
/// 1) Stores all the nodes in memory, so please retain from using it if  
///    the available amount of memory is limited compared to the number  
///    of leaves to be stored;  
/// 2) Leaves and nodes are hashed without using any kind of domain separation:  
///    while this is ok for use cases where the Merkle Trees have always the  
///    same height, it's not for all the others.
```

When Merkle tree implementations do not intrinsically differentiate between *internal* nodes and *leaf* nodes when hashing them, they may lack *second-preimage resistance*: given a root R and tree T , it is possible to compute a tree T' that also produces¹¹ R .

A trivial demonstration of this weakness is depicted in the two figures below. Consider the Merkle tree built with the four leaves L_1, L_2, L_3, L_4 , where the values of the internal node N_1 is $N_1 = H(L_1, L_2)$ and $N_2 = H(L_3, L_4)$ and the resulting root R is $R = H(N_1, N_2)$, as depicted in Figure 1.

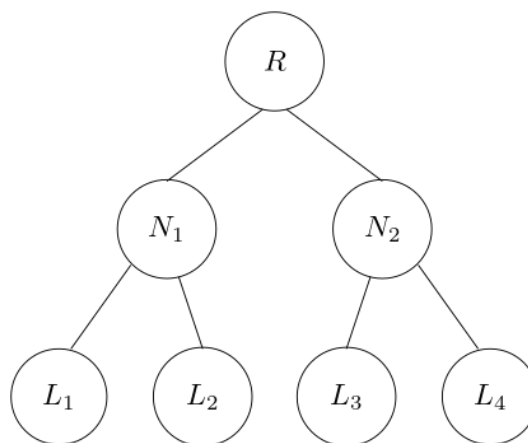


Figure 1: Merkle Tree with four leaves

10. <https://github.com/HorizenOfficial/ginger-lib/issues/110>

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



It can be seen that a tree created with the two values N_1 and N_2 as leaves will result in the same root value, as depicted in Figure 2.

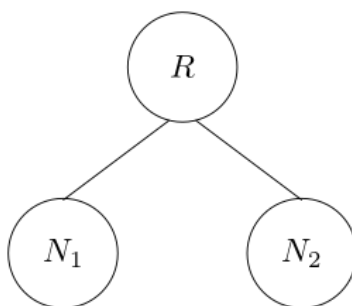


Figure 2: Merkle Tree with two leaves, resulting in the same root

While slightly contrived, this example shows that second-preimage resistance is not fulfilled when trees do not differentiate between leaves and internal nodes for hashing. A more concrete attack was described for Bitcoin, in which an attacker could perform a series of brute-force attacks in order to craft a 64-byte transaction that is submitted to the Bitcoin blockchain, and this transaction would allow them to prove inclusion of a rogue transaction which was never included in the Bitcoin blockchain. A blog post by Sergio Damian Lerner¹² explores this attack in detail: the cost is that of brute-forcing a relatively large search space (between 69 and 73 bits).

Recommendation

Consider adding a domain separator tag to the oracle calls, different for the intermediate hash and leaf hash calls.

Location

- [ginger-lib/primitives/src/merkle_tree/field_based_mht/append_only/mod.rs](#)
- [ginger-lib/primitives/src/merkle_tree/field_based_mht/naive/mod.rs](#)
- [ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/big_lazy_merkle_tree.rs](#)

Retest Results

2023-03-06 – Not Fixed

Since the current implementation only uses trees with a fixed height, the Horizen Labs team deemed this issue to be non-exploitable.

However, since *ginger-lib* aims to be a generic library, the team opened an [issue](#) to track this finding with the intent of eventually fixing it.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

All the trees we are using are of fixed height so we are not exposed to second preimage attacks.

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



Missing Enforcement of Minimum Seed Size in KeyGen

Overall Risk Medium

Impact High

Exploitability Medium

Finding ID NCC-E005939-XX3

Component zendoo-sc-cryptolib

Category Cryptography

Status Risk Accepted

Impact

Users may inadvertently use trivial-sized seeds to generate weak keys easily susceptible to brute-force attacks.

Description

The `schnorr_derive_key_from_seed()` function implemented in the `cctp_calls.rs` source file derives a key from a provided seed. Other than using a random number generator when an empty seed is provided, there is no validation of the provided seed's size. A portion of this function is excerpted below.

```
90 pub fn schnorr_derive_key_from_seed(seed: &[u8]) -> (SchnorrPk, SchnorrSk) {
91     // zero just default to random,
92     // however, is there a minimum length that should be required?
93     if seed.is_empty() {
94         return schnorr_generate_key();
95     }
96
97     // Domain separation tag
98     const DST: &[u8] = &[0xFFu8; 32];
99
100    // Hash first to ensure size an eliminate any bias
101    // that may exist in `seed`
102    let mut hasher = blake2::Blake2b::default();
103    hasher.input(DST);
104    hasher.input(seed);
105    let digest = hasher.fixed_result();
106    ...
```

The highlighted comment above is correct that a minimum size should be enforced here. This function is exposed in the Java JNI code¹³ without any size validation present.

Another instance is present within the `vrf_derive_key_from_seed()` function implemented in the same source file.

Recommendation

Verify that seeds provided to all keygen functions satisfy a minimum size, ideally of comparable size to the curve order.

Location

`zendoo-sc-cryptolib/api/src/cctp_calls.rs`

13. <https://github.com/HorizenOfficial/zendoo-sc-cryptolib/blob/2007463981fb39a7aa98f742c8cd75df03013097/api/src/lib.rs#L536>



Retest Results

2023-03-06 – Not Fixed

The Horizen Labs team decided to maintain the functionality of the key derivation functions to use an arbitrarily short seed and essentially shift the responsibility to the caller of these functions.

As such, in [commit 8f2cb20](#), the team added documentation to the functions `schnorr_derive_key_from_seed()` and `vrf_derive_key_from_seed()` indicating that it was the caller's responsibility to pass a seed of proper length. Additionally, the Horizen Labs team added warnings to the documentation of the `generate()` functions of the Java classes `SchnorrKeyPair` and `VRFKeyPair`.

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

It is more convenient for us to handle this directly on the SDK side.

The SDK will take care of passing an appropriate seed to the Rust functions.

In order to make this explicit, we added comments in the code with [commit 8f2cb20](#).



Problematic Point at Infinity Representation and Comparison

Overall Risk Medium

Impact Medium

Exploitability Low

Finding ID NCC-E005939-HWH

Component ginger-lib

Category Cryptography

Status Fixed

Impact

Arbitrary representations of the point at infinity, as well as missing checks for this special case in the equality function, may introduce subtle vulnerabilities in cryptographic operations.

Description

The file [short_weierstrass_projective.rs](#)¹⁴ defines the representation of an elliptic curve point in affine coordinates. In addition to the `x` and `y` coordinates, the structure also specifies a boolean, `infinity`, defining whether the point is the neutral element (i.e., the point at infinity). This can be seen in the code excerpt below.

```
pub struct AffineGadget<
    P: SWModelParameters,
    ConstraintF: PrimeField,
    F: FieldGadget<P::BaseField, ConstraintF>,
> {
    pub x: F,
    pub y: F,
    pub infinity: Boolean,
    _params: PhantomData<P>,
    _engine: PhantomData<ConstraintF>,
}
```

Checking that an element is the neutral element can be performed by calling the `is_zero()` function (copied below). This function simply returns the value of the boolean `infinity`. Hence, a point with arbitrary values for its `x` and `y` coordinates will still be interpreted as the point at infinity, as long as `self.infinity` is set to true. This essentially allows an arbitrarily large number of representations of that special element.

```
#[inline]
fn is_zero<CS: ConstraintSystemAbstract<ConstraintF>>(
    &self,
    _: CS,
) -> Result<Boolean, SynthesisError> {
    Ok(self.infinity)
}
```

14. Respectively [short_weierstrass_jacobian.rs](#), since the two files are essentially the same.



As a side observation, note that the neutral element is defined as the point with the coordinates $(0, 1)$ and the `infinity` flag set to true, at least by default. This can be seen in the `zero()` function provided below, for reference.

```
#[inline]
fn zero<CS: ConstraintSystemAbstract<ConstraintF>>(mut cs: CS) -> Result<Self,
↳ SynthesisError> {
    Ok(Self::new(
        F::zero(cs.ns(|| "zero"))?,
        F::one(cs.ns(|| "one"))?,
        Boolean::constant(true),
    ))
}
```

This ambiguous definition of the point at infinity may become problematic in a number of cases, such as when comparing two `AffineGadget`s with the `eq()` function, which only compares the values of the `x` and `y` coordinates, but fails to check whether the point is the neutral element (as dictated by the `infinity` flag).

```
impl<P, ConstraintF, F> PartialEq for AffineGadget<P, ConstraintF, F>
where
    P: SWModelParameters,
    ConstraintF: PrimeField,
    F: FieldGadget<P::BaseField, ConstraintF>,
{
    fn eq(&self, other: &Self) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```

As such, the function above may consider two points to be equal, even if one is the point at infinity, or alternatively, it may consider two representations of the point at infinity to be different (even if the function `is_zero()` would evaluate to true for both instantiations).

Recommendation

Update the `eq()` function to correctly perform equality checking of the neutral element. Inspiration could be taken from the `is_eq()` function in the same file, which correctly handles the case of the point at infinity. Additionally, consider enforcing a single representation for the point at infinity, which would presumably limit ambiguities and potential for issues such as the one described in this finding. A step towards this direction could be to force the encoding functions defined in this file to encode the neutral element as its default value (as defined in the `zero()` function), and not to preserve the value of arbitrary coordinates.

Consider additionally deleting one of the two files `short_weierstrass_projective.rs` and `short_weierstrass_jacobian.rs` and renaming it to a more fitting name.

Location

- [ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_projective.rs](#)
- [ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_jacobian.rs](#)



Retest Results

2023-03-03 – Fixed

In response to this finding and the related [finding "Incomplete Elliptic Curve Formulas"](#), the Horizen Labs team introduced changes in two pull requests, which address the issues outlined in this finding. Specifically, [pull request 198](#) for *ginger-lib* updated the code in the following ways.

1. A number of checks have been introduced, ensuring that neither point is the neutral element in the functions `add_internal()` and `double_and_add_internal()`, and ensuring that `self` is not the point at infinity in the functions `double_in_place()` and `add_constant()`. This also ensures the higher-level functions `add()` and `double_and_add()` are safely guarded.
2. The implementation now enforces a unique representation for the point at infinity, namely `(x = 0, y = 1, infinity = true)`.
3. The missing infinity check in the `PartialEq` implementation was fixed.

The second pull request, [PR 116](#) for *zendoo-sc-cryptolib*, refactored some code to reflect the changes introduced in *ginger-lib*.

The second and third items of the list above are aligned with NCC Group's recommendation for this finding. This finding is considered *fixed* as a result.



Non-Injective `encode_to_curve` Violates VRF Collision Resistance Assurance

Overall Risk Medium

Impact High

Exploitability Low

Finding ID NCC-E005939-HHX

Component ginger-lib

Category Cryptography

Status Fixed

Impact

A non-injective `encode_to_curve` function will cause the VRF to violate the collision resistance assurance required to avoid a proof corresponding to multiple values.

Description

The CFRG VRF specification outlines an assurance of collision resistance as the infeasibility of finding two input messages/values (`alpha`) with the same output (`beta`). The functional path of interest involves encoding `alpha` to a curve point, multiplying that curve point by the secret key, and hashing the result to obtain `beta`. As described below, the VRF `encode_to_curve` process may not be injective, so two messages (`alpha1` and `alpha2`) may map to the same encoded curve point and ultimately result in the same output `beta`. This would violate the VRF assurance of collision resistance. The intermediate calculation of the full proof (`pi_string`) elements `c` and `s` is not relevant here.

In the `ginger-lib/primitives/src/vrf/ecvrf/mod.rs` source file, both the `prove()` and `proof_to_hash()` functions call the `evaluate()` function to encode the message into a curve point as excerpted below.

```
329 //Compute mh = hash_to_curve(message)
330 let message_on_curve =
331     GH::evaluate(group_hash_params, to_bytes!(&message).unwrap().as_slice())?;
```

Prior to this message-to-point encoding, the `setup()` function in the `ginger-lib/primitives/src/crh/bowe_hopwood/mod.rs` source file is run to calculate the maximum number of chunks in a segment via the helper function excerpted below, which returns 127 for the Tweedle curves.

```
51 fn calculate_num_chunks_in_segment<F: PrimeField>() -> usize {
52     let upper_limit = F::modulus_minus_one_div_two();
53     let mut c = 0;
54     let mut range = F::BigInt::from(2_u64);
55     while range < upper_limit {
56         range.muln(4);
57         c += 1;
58     }
59
60     c
61 }
```

The above calculated value `c` is next validated against the `WINDOW_SIZE` to ensure the `evaluate()` function cannot overflow and potentially generate a collision. However, section 5.4.1.7 of the Zcash Protocol Specification¹⁵ specifies a different constraint on the `c` value from that produced by the above helper function:



$$4 * (2^{4*c} - 1) / 15 \leq (r - 1) / 2, \text{ i.e. } c = 63 \text{ (for Tweedle)}$$

As such, the `WINDOW_SIZE` of 128 or 90 that is set in a variety of source files (and is appropriate for the prior MNT curves) will produce a non-injective `evaluate()` function for message-to-point encoding under the Tweedle curves. These collisions will cause the VRF to violate its assurance of collision resistance.

Separately, note that section 5.4.1.7 of the Zcash Protocol Specification describes the collision resistance of the Pedersen hash function in the context of a fixed-length input. However, the `evaluate()` function takes an input slice of arbitrary length and validates the input length against a maximum size as excerpted below. There is no minimum size check.

```
84 fn evaluate(parameters: &Self::Parameters, input: &[u8]) -> Result<Self::Output, Error> {
85     let eval_time = start_timer!(|| "BoweHopwoodPedersenCRH::Eval");
86
87     if (input.len() * 8) > W::WINDOW_SIZE * W::NUM_WINDOWS * CHUNK_SIZE {
88         return Err(Box::new(CryptoError::Other(
89             format!(
90                 "incorrect input length {:?} for window params {:?}x{:?}x{}",
91                 input.len(),
92                 W::WINDOW_SIZE,
93                 W::NUM_WINDOWS,
94                 CHUNK_SIZE,
95             )
96             .to_owned(),
97         )));
98     }
99     ...
```

While the initial concern around length constraints likely pertain to overflows as described above, allowing shorter lengths will also give significant advantage to an attacker. Horizon has indicated to NCC Group that this function is always called with fixed size input. Note that Zcash has explicitly added the following note to their protocol specification on page 78.

Non-normative note: These hash functions are not collision-resistant for variable-length inputs.

Recommendation

Correct the functionality of the `calculate_num_chunks_in_segment()` function to reflect the limits shown in the Zcash Protocol Specification constraint. Accordingly, set the `WINDOW_SIZE` to 63 or less.

Implement an equality length check of the `evaluate()` function input on the highlighted line 87 above (for a fixed value smaller than the calculated maximum constant). Alternatively, adapt the function signature to accept a fixed array of correct length. Ensure all calling functions respect the fixed length input constraint.

Location

- [ginger-lib/primitives/src/vrf/ecvrf/mod.rs](#)
- [ginger-lib/primitives/src/crh/bowe_hopwood/mod.rs](#)

15. <https://zips.z.cash/protocol/protocol.pdf>



Retest Results

2023-03-02 – Fixed

In response to this finding, the Horizen Labs team introduced changes addressing the issues outlined in this finding.

- [Commit 84cba3f9](#) for *zendoo-sc-cryptolib* updates the VRF parameter from 128 to 63.
- [Pull request 196](#) for *ginger-lib* adds explicit checks in the implementations of `FixedLengthCRH`. These various checks now ensure that the generic parameters are correct (for example via the function `check_preconditions()` in *primitives/src/crh/bowe_hopwood/mod.rs*) and that the input to the Pedersen hash function is of expected size.

These changes are in line with the recommendation above, and this finding is considered *fixed* as a result.



Incomplete Elliptic Curve Formulas

Overall Risk Medium
Impact High
Exploitability Medium

Finding ID NCC-E005939-39X
Component ginger-lib
Category Cryptography
Status Fixed

Impact

The lack of complete addition formulas can lead to unexpected issues in case adversaries are given the possibility to use carefully crafted points in curve point operations. Additionally, if applications build custom cryptographic protocols leveraging these operations, then this issue may allow bypassing such protocols.

Description

In elliptic curve cryptography, the addition law of points on the curve is said to be *complete* if it correctly computes the sum of *any* two points in the group, including all the special cases (namely, when one of the two operands is the point at infinity, when one operand is the inverse of the other one, or when the two operands are equal).

The addition law defined in the file [short_weierstrass_projective.rs](#)¹⁶ in the function `add_internal()` is incomplete, as also documented in the comments preceding the function implementation. The signature of this function and related comments are excerpted below, for reference.

```
/// Incomplete addition: neither `self` nor `other` can be the neutral
/// element, and other != ±self.
/// If `safe` is set, enforce in the circuit exceptional cases not occurring.
fn add_internal<CS: ConstraintSystemAbstract<ConstraintF>>(<
    &self,
    mut cs: CS,
    other: &Self,
    safe: bool,
)> -> Result<Self, SynthesisError> {
    //<snip>
```

A few comments can be made about that function and the general lack of support for special operands. Firstly, the meaning and the description of the `safe` flag is slightly misleading. Namely, setting the `safe` parameter to `true` does not mean that the function correctly handles special cases; it merely adds a constraint in the constraint system that enforces that the `x`-coordinates of both points are different (covering both point doubling and addition of a point with its inverse). However, the `safe` parameter does not *explicitly* guard against bad usage. Specifically, it does not resort to doubling in case the points are equal, nor does it correctly handle the case of the neutral element. Further, the special case of the point at infinity will be silently processed, since it also admits a representation including `x` and `y` coordinates, as discussed in [Finding "Problematic Point at Infinity Representation and Comparison"](#).

Secondly, no functions (or combination thereof) in this file support general point addition covering all the special cases mentioned above. A `double_in_place()` function is defined (which covers the case of adding a point to itself), but there currently is not any function

16. Respectively [short_weierstrass_jacobian.rs](#), since the two files are essentially the same.



that can be used to perform $P + (-P)$, or $P + 0$. Even though these cases may seem rare, they do occur in practice. For example, a common procedure to check if a point is in the correct subgroup is to check if the point multiplied by the subgroup order (call it n) is equal to the neutral element. That is, compute and check whether $nP = 0$. The lack of complete addition formulas means that given the current algorithms, it seems impossible to perform a computation like nP , since the computation would not result in the neutral element.

Generally, the lack of complete addition formulas can lead to unexpectedly complex and subtle attacks, particularly if malicious participants were able to selectively provide points to be used in arithmetic operations. Even in non-malicious scenarios, users of the library may call these functions with certain expectations in mind, which could lead to breaks of their protocols.

Additionally, note that the two functions `double_and_add_internal()` and `add_constant()` also suffer from the issues described in this finding.

In comparison, consider Zcash's Elliptic Curve Gadget implementation for Halo2, located in the file `halo2_gadgets/src/ecc.rs`. In this implementation, the default `add()` function performs *complete point addition* while another function, `add_incomplete()`, makes it explicit that it does not handle special cases. The signatures of these two functions are provided below, for reference.

```
/// Performs incomplete point addition, returning `a + b`.
///
/// This returns an error in exceptional cases.
fn add_incomplete(
    &self,
    layouter: &mut impl Layouter<C::Base>,
    a: &Self::NonIdentityPoint,
    b: &Self::NonIdentityPoint,
) -> Result<Self::NonIdentityPoint, Error>;

/// Performs complete point addition, returning `a + b`.
fn add<A: Into<Self::Point> + Clone, B: Into<Self::Point> + Clone>(
    &self,
    layouter: &mut impl Layouter<C::Base>,
    a: &A,
    b: &B,
) -> Result<Self::Point, Error>;
```

Recommendation

Consider implementing complete addition formulas allowing users to perform all desired operations on curve points. The addition of validation checks ensuring no operand is the neutral element could also be introduced in these functions. The naming of these functions could follow Zcash's example, making the default addition operation the complete one.

At the very least, precisely document what the different functions described in this finding do and what limitations they have with respect to the value of their operands.

Location

- `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_projective.rs`
- `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_jacobian.rs`



Retest Results

2023-03-03 – Fixed

In response to this finding and the related [finding "Problematic Point at Infinity Representation and Comparison"](#), the Horizen Labs team introduced changes in two pull requests, which address the issues outlined in this finding. Specifically, [PR 198](#) for *ginger-lib* updated the code in the following ways.

1. A number of checks have been introduced, ensuring that neither point is the neutral element in the functions `add_internal()` and `double_and_add_internal()`, and ensuring that `self` is not the point at infinity in the functions `double_in_place()` and `add_constant()`. This also ensures the higher-level functions `add()` and `double_and_add()` are safely guarded.
2. The implementation now enforces a unique representation for the point at infinity, namely `(x = 0, y = 1, infinity = true)`.
3. The missing infinity check in the `PartialEq` implementation was fixed.

The implementation of complete addition formulas was postponed for the time being as it is not deemed necessary (see the section *Client Response* below).

The second pull request, [PR 116](#) for *zendoo-sc-cryptolib*, refactored some code to reflect the changes introduced in *ginger-lib*.

The first item of the list above is aligned with NCC Group's recommendation for this finding. The Horizen Labs team also provided sensible reasoning for the decision to postpone the implementation of the other recommendations. This finding is considered *fixed* as a result.

Client Response

Regarding complete arithmetic, it has not been necessary for our circuits so far, so we are evaluating whether to postpone its introduction in *ginger-lib* to future releases in case we will need it in our circuits.



Insecure Public Keys or Signatures can be Generated for Schnorr Signatures

Overall Risk Low

Impact High

Exploitability Low

Finding ID NCC-E005939-XKT

Component ginger-lib

Category Cryptography

Status Fixed

Impact

If a Schnorr signature is generated with a random value of `k = 0`, the resultant signature would leak the secret key. Similarly, if a Schnorr public key is generated with a `secret_key = 0`, the public key will simply be the identity element, which allows signatures to be trivially forgeable.

Description

The function `FieldBasedSchnorrSignatureScheme::sign()` in `primitives/src/signature/schnorr/field_based_schnorr.rs` generates a truncated Schnorr Signature. As part of this process, a random field element `k` is generated as follows:

```
245 //Sample random element
246 let k = G::ScalarField::rand(rng);
```

The signature is then computed as $(s, e) = (k - sk * e, H(g^k || m))$. However, Schnorr signatures require the random value `k` to be generated within the range $[1, q-1]$, where `q` denotes the size of the scalar field. In particular, the value `k = 0` is disallowed, as it would leak the secret key. This is not checked in `FieldBasedSchnorrSignatureScheme::sign()`, possibly resulting in insecure signatures.

Similarly, the functions `SchnorrSignature::sign()` and `SchnorrSignature::keygen()` do not verify that `k != 0` and `secret_key != 0` respectively, but do not appear to be used anywhere (the function `FieldBasedSchnorrSignatureScheme::keygen()` performs this check correctly).

Recommendation

Modify the signing function to check that `k != 0` before proceeding. This is already done, for example, during `keygen()` for truncated Schnorr Signatures:

```
fn keygen<R: Rng>(rng: &mut R) -> (Self::PublicKey, Self::SecretKey) {
    let secret_key = loop {
        let r = G::ScalarField::rand(rng);
        // Reject sk = 0 to avoid generating obviously weak keypair. See keyverify() function
        // for additional explanations.
        if !r.is_zero() {
            break (r);
        }
    };
};
```

Location

- `ginger-lib/primitives/src/signature/schnorr/field_based_schnorr.rs:L246`
- `ginger-lib/primitives/src/signature/schnorr/mod.rs:L93`



-
- [ginger-lib/primitives/src/signature/schnorr/mod.rs:L110](#)

Retest Results

2023-03-02 – Fixed

In [pull request 183](#), the Horizen Labs team added checks ensuring the randomly generated Schnorr nonce is not zero, as outlined in the Recommendation section above.

As a result, this finding is considered *fixed*.



Silent `pow()` Overflow in Merkle width and height

Overall Risk	Low	Finding ID	NCC-E005939-9HX
Impact	Medium	Component	ginger-lib
Exploitability	Undetermined	Category	Data Validation
		Status	Fixed

Impact

Accidental misuse of the `height` parameter when calling `init()` may result in a silent overflow during the calculation of the `width` value (stemming from an unchecked `pow()`).

Description

The `init()` function from `ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/big_lazy_merkle_tree.rs` is excerpted below.

```

49 pub fn init(height: u8) -> Self {
50     assert!(check_precomputed_parameters:::<T>(height as usize));
51
52     let rate = <<T::H as FieldBasedHash>::Parameters as FieldBasedHashParameters>::R;
53
54     assert_eq!(T::MERKLE_ARITY, 2); // For now we support only arity 2
55     // Rate may also be smaller than the arity actually, but this assertion
56     // is reasonable and simplify the design.
57     assert_eq!(rate, T::MERKLE_ARITY);
58
59     // If height is 0 it must not be possible to add any leaf, so we'll set width to 0.
60     let width: u32 = if height != 0 {
61         T::MERKLE_ARITY.pow(height as u32) as u32
62     } else {
63         0
64     };

```

On the highlighted line 61 above, if the user passes a `height` parameter of 32 or more into the `init()` function, the `pow()` function will silently overflow in release mode and program execution will continue with a corrupted configuration.

A similar issue is present on lines 63, 337, and 347 of `append_only/mod.rs` with a `pow()` that can silently overflow.

As an aside, note that the `height` parameter is of type `u8` in `big_lazy_merkle_tree.rs`, while the `height` parameter is of type `usize` in the naive and append-only versions. In the latter version, the initial/final/processed `position` vectors contain the type `u32` which requires a lot of downstream conversions. Note that the definition of `usize` changes depending upon the target architecture, and that the JNI code utilizes `_height` as an `i32`¹⁷.

There are two related instances of `pow()` involving `height` in `zendoo-sc-cryptolib` (see location below).

17. <https://github.com/HorizenOfficial/zendoo-sc-cryptolib/blob/2007463981fb39a7aa98f742c8cd75df03013097/api/src/lib.rs#L1193>



Recommendation

Replace `pow()` with `checked_pow()` and handle the error condition similarly to the preceding `assert_eq` statements. Favor consistent fixed-size types when possible.

Location

- [ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/big_lazy_merkle_tree.rs](#)
- [ginger-lib/primitives/src/merkle_tree/field_based_mht/append_only/mod.rs](#)
- [zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/constraints/data_structures.rs:L26,L133](#)

Retest Results

2023-03-03 – Fixed

In [pull request 191](#) for *ginger-lib*, and [pull request 113](#) for *zendoo-sc-cryptolib*, the Horizen Labs team replaced the vulnerable instances of the function `pow()` identified in this finding with `checked_pow()`. Appropriate errors are now also returned when overflows occur.

This finding is considered *fixed* as a result.



Confusing Endianness in UInt8 Vector Creation

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E005939-LUC

Component ginger-lib

Category Cryptography

Status Fixed

Impact

Unexpected or uncommon endianness representations may impact correctness of cryptographic operations and lead to more subtle and unexpected vulnerabilities.

Description

The file `uint8.rs` in `r1cs/gadgets/std/src/bits/` defines the `UInt8` structure, a representation of an unsigned 8-bit integer as a vector of 8 `Boolean` objects, to be used by the Rank-1 Constraint System (R1CS). The `UInt8` type defines the function `constant_vec()`, which is used to create a vector of `UInt8`s from a vector of `u8`s, the latter being the Rust primitive unsigned 8-bit integer type. This function relies on the underlying `constant()` function, which creates a `UInt8` from a single Rust `u8`. Both functions are excerpted below, for reference.

```
/// Construct a constant vector of `UInt8` from a vector of `u8`
pub fn constant_vec(values: &[u8]) -> Vec<Self> {
    let mut result = Vec::new();
    for value in values {
        result.push(UInt8::constant(*value));
    }
    result
}

/// Construct a constant `UInt8` from a `u8`
pub fn constant(value: u8) -> Self {
    let mut bits = Vec::with_capacity(8);

    let mut tmp = value;
    for _ in 0..8 {
        // If last bit is one, push one.
        if tmp & 1 == 1 {
            bits.push(Boolean::constant(true))
        } else {
            bits.push(Boolean::constant(false))
        }

        tmp >>= 1;
    }

    Self {
        bits,
        value: Some(value),
    }
}
```



The output of the conversion performed by the `constant_vec()` function results in an unusual endianness format. Specifically, the `constant()` function performs the conversion by flipping the bits of their byte representations. As such, a (big-endian-represented) byte will be converted to its *little-endian* bit representation. The use of the `constant()` function by the `constant_vec()` function induces additional confusion, since that function creates a vector of `UInt8`s from a vector of `u8`s by maintaining their byte ordering. Hence, feeding a big-endian byte array into that function will result in a byte array with the bytes arranged in the same order, but with their individual bits reversed.

As an example, calling the `constant_vec()` function with the input `vec![0x01, 0x0F]` results in a vector whose bit-representation is the following:

```
[true, false, false, false, false, false, false, false], [true, true, true, true, false, false, false, false]]
```

Which, re-interpreted as bytes, would be `[0x80, 0xF0]`.

The section [Non-Security Impacting Observations](#) already highlighted the usage of three different endianness types within the code base under review (*little-endian byte*, *big-endian bit*, and *little-endian bit*), to which this finding adds a fourth, less common and potentially more problematic one.

The function `constant_vec()` is currently only used by a test functionality in `r1cs/gadgets/crypto/src/merkle_tree/mod.rs`, thereby limiting the impact of this finding.

Recommendation

Carefully assess what endianness format shall be used by the gadgets and ensure all bytes and bit representations are consistent (preferably little- or big-endian representations at a *byte* level). Update the `constant_vec()` function to follow that convention, or consider deleting it in case it is not used by any other function.

Location

- [ginger-lib/r1cs/gadgets/std/src/bits/uint8.rs](#)

Retest Results

2023-03-06 – Fixed

In [pull request 193](#), the Horizen Labs team removed the `constant_vec()` function from the code base as it was only used by one test, thereby preventing future misuses of that function.

This finding is considered *fixed* as a result.



Panic in Field Element Gadget Deserialization from Empty Array

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E005939-74E

Component ginger-lib

Category Data Validation

Status Fixed

Impact

Missing parameter checks resulting in a panic may present a denial-of-service attack vector.

Description

The function `from_bits()` located in `r1cs/gadgets/std/src/fields/fp.rs` converts a slice of `Boolean` gadgets into a field element, represented as an `FpGadget`. An excerpt of the first few lines of that function can be seen below.

The NCC Group consultants noted that a panic can be triggered when the bit-slice received as parameter is empty. Specifically, the call to `unwrap()` highlighted in the snippet below will cause a panic since there is no element to iterate over.

```
// Pack a slice of Boolean gadgets into one or several field elements, bundling at most
// `F::Params::CAPACITY` many in a single field element (to allow efficient unpacking).
// The bundling regards the Booleans in big endian order.
impl<F: PrimeField> FromBitsGadget<F> for FpGadget<F> {
    fn from_bits<CS: ConstraintSystemAbstract<F>>>(
        mut cs: CS,
        bits: &[Boolean],
    ) -> Result<Self, SynthesisError> {
        // We can safely pack up to CAPACITY bits
        let bits = bits.chunks(F::Params::CAPACITY as usize).next().unwrap();

        let mut num = Self::zero(cs.ns(|| "alloc_lc_{}"))?;
        let mut coeff = F::one();

        // <snip>
```

Additionally, the documentation comments preceding the function are incorrect. They state that the function packs `Boolean`s into one or several field elements, while this function actually only packs them into one, discarding the superfluous bits.

Recommendation

Modify the `from_bits()` function to return an error in case the `bits` parameter is empty and update the function documentation to reflect the fact that only one element will be created. Additionally, consider returning an error in case the length of the `bits` parameter exceeds `F::Params::CAPACITY`.

Location

[ginger-lib/r1cs/gadgets/std/src/fields/fp.rs:L496-L497](#)



Retest Results

2023-03-03 – Fixed

In [pull request 191](#), the Horizen Labs team added some checks in the `from_bits()` function, which now returns errors in case the bit-slice is empty or is larger than the capacity.

The function documentation has also been updated in accordance to the recommendation above.

As such, this finding is considered *fixed*.



Hashing to Elliptic Curve is Biased and Not Standard

Overall Risk Informational

Impact None

Exploitability None

Finding ID NCC-E005939-6AH

Component zendoo-sc-cryptolib

Category Cryptography

Status Risk Accepted

Impact

The hash-to-curve implementation used to compute the *phantom key* pk_{NULL} does not implement a proper hash-to-curve algorithm. In particular, the distribution from which the phantom key is sampled is biased, which could lead to a degradation in security.

Description

The *phantom key* pk_{NULL} is a randomly chosen fixed element from the group used as a placeholder public key for threshold signatures, to which nobody knows the secret key. The accompanying [documentation](#) describes a process for choosing such a key as follows: “choosing it as hash of some public data, for example $pk_{\text{NULL}} = H(\text{“magic string”})$ where H is any hash-to-curve algorithm”.

This is implemented by the `hash_to_curve()` function defined in `demo-circuit/src/constants/mod.rs`, and the output is saved in the `NaiveThresholdSigParams` structure to avoid recomputing it every time. (A similar process is also followed for the `VRFParams` structure.) While *any* hash-to-curve algorithm is permitted, note that the implemented algorithm differs from a standard hash-to-curve implementation, such as the one described in the [IRTF hash-to-curve draft](#) in several respects:

1. The function implemented is closer to the `map_to_curve` function described in the draft than to the `hash_to_curve` one. In particular, the `map_to_curve` map does not have a uniform output. In order to obtain an output distribution indistinguishable from a uniform selection over the whole curve, the `hash_to_curve` function from [draft-irtf-cfrg-hash-to-curve](#) performs two mappings of hash-derived data into curve points: the original input string is hashed twice (along with disambiguating tags), yielding two strings that are mapped to two curve points, which are then added together. The `hash_to_curve()` function in `demo-circuit/src/constants/mod.rs` implements only a single map.
2. The map implemented is not equivalent to the [map-to-curve function](#) from the hash-to-curve draft. In particular, for an elliptic curve defined as $y^2 = g(x)$, this mapping does not succeed if the given field element x does not result in a square $g(x)$. This happens with probability around $1/2$ or so; in practice, the mapping will need to be run at least twice. The hash-to-curve draft instead describes a deterministic map from an input u to an element x with $g(x)$ square.

Recommendation

Modify the phantom key generation procedure to follow the process described in the hash-to-curve draft, to avoid any potential biases in the selection of the placeholder public key.

Location

[zendoo-sc-cryptolib/demo-circuit/src/constants/mod.rs:L258](#)



Retest Results

2023-03-02 – Not Fixed

With [pull request 101](#), the Horizen Labs team added the following inline documentation warning users about the use of that function for generic purposes.

```
// This method is safe to use only to generate fixed curve elements in a deterministic  
// fashion. It should not be employed as a method to hash a sequence of bytes to a  
// curve element, as it does not fulfill all the necessary statistical requirements of a  
// standard-compliant hash function to curve elements
```

The status of this finding was updated to *Risk Accepted* as a result.

Client Response

Though we acknowledged the issue reported with our hash-to-curve implementation being non standard compliant, we have decided to avoid modifications to our implementation for the following reasons:

- The same implementation is employed without issues so far in ZCash
- We use it only to generate constants for our circuits and show that they were generated following a deterministic process



Outdated Rust Toolchain and Dependencies

Overall Risk Informational

Impact None

Exploitability None

Finding ID NCC-E005939-9XW

Component Systemic

Category Patching

Status Partially Fixed

Impact

Attackers may attempt to identify and utilize publicly known vulnerabilities in outdated dependencies to exploit the functionality of the target code.

Description

Incorporating outdated dependencies is one of the most common, serious, and exploited application vulnerabilities. Inconsistent toolchains can also increase the difficulty to build and debug projects.

The Rust toolchain specified in the `rust-toolchain` file in both `zendoo-sc-cryptolib` and `ginger-lib` is version `1.51.0`, which is almost two years out of date¹⁸. The latest version is `Version 1.67.0` dated `2023-01-26`.

The two libraries under review also use many dependencies that are marginally out of date. The `cargo-outdated` tool can be used to check for such out-of-date dependencies. Since the projects rely on a significant number of dependencies, and the tool is quite verbose, its output was not directly copied in this finding. However, running `cargo outdated` on the `ginger-lib` project results in close to 400 results (many of which are duplicates), see below.

```
~/ginger-lib (development)> cargo outdated | wc -l
404
```

In addition to `cargo-outdated`, the `cargo-audit` tool is useful for assessing known vulnerabilities in existing dependencies. Running this tool on the `zendoo-sc-cryptolib` repository outputs the following vulnerability in the `bzip2` crate.

```
Crate:      bzip2
Version:    0.4.3
Title:      bzip2 Denial of Service (DoS)
Date:       2023-01-09
ID:         RUSTSEC-2023-0004
URL:        https://rustsec.org/advisories/RUSTSEC-2023-0004
Solution:   Upgrade to >=0.4.4
Dependency tree:
bzip2 0.4.3
├─ cctp_primitives 0.1.2
│   └─ demo-circuit 0.6.0-SNAPSHOT
│       └─ api 0.6.0-SNAPSHOT
│           └─ api 0.6.0-SNAPSHOT
error: 1 vulnerability found!
```

18. <https://blog.rust-lang.org/2021/03/25/Rust-1.51.0.html>



Similarly, running this tool on the *ginger-lib* repository produces the following two warnings.

```
Crate:    serde_cbor
Version:  0.11.2
Warning:  unmaintained
Title:    serde_cbor is unmaintained
Date:     2021-08-15
ID:       RUSTSEC-2021-0127
URL:      https://rustsec.org/advisories/RUSTSEC-2021-0127
Dependency tree:
serde_cbor 0.11.2
└─ criterion 0.3.5
   └─ proof-systems 0.4.0
      └─ r1cs-crypto 0.4.0
         └─ r1cs-crypto 0.4.0
            └─ proof-systems 0.4.0
               // <snip>

Crate:    term
Version:  0.5.2
Warning:  unmaintained
Title:    term is looking for a new maintainer
Date:     2018-11-19
ID:       RUSTSEC-2018-0015
URL:      https://rustsec.org/advisories/RUSTSEC-2018-0015
Dependency tree:
term 0.5.2
└─ clippy 0.0.302
   └─ algebra 0.4.0
      └─ r1cs-std 0.4.0
         └─ r1cs-crypto 0.4.0
            └─ r1cs-crypto 0.4.0
               └─ proof-systems 0.4.0
                  └─ r1cs-crypto 0.4.0
                     // <snip>

warning: 2 allowed warnings found
```

Recommendation

Update the `rust-toolchain` files to the latest version suitable for deployment. Add a periodic gating milestone to the project plan that involves reviewing all dependencies for outdated or vulnerable versions.

Location

- [zendoo-sc-cryptolib/rust-toolchain](#)
- [ginger-lib/rust-toolchain](#)

Retest Results

2023-03-07 – Partially Fixed

In a set of three pull requests (namely, [PR 105](#) and [PR 107](#) for *zendoo-sc-cryptolib* and [PR 189](#) for *ginger-lib*), the Horizen Labs team updated a number of dependencies and introduced changes to `cargo-audit` usage to catch advisories with informational warnings set to *unsound* and *unmaintained*. Additionally, the vulnerable `bzip2` dependency identified above was updated to version 0.4.4.



Since the toolchain version was not updated, the status of this informational finding was set to *Partially Fixed*.



VRF Implementation Anomalies

Overall Risk	Informational	Finding ID	NCC-E005939-PW9
Impact	Undetermined	Component	ginger-lib
Exploitability	Undetermined	Category	Cryptography
		Status	Fixed

Impact

Minor informational anomalies may become significant within a larger/changed context.

Description

This informational-only finding documents several anomalies found while reviewing the VRF implementation. There is no current security implication.

Within *ginger-lib/primitives/src/vrf/ecvrf/mod.rs*¹⁹ it was observed that:

1. Line 239 demonstrates that sampling a random field element can return `0` (rather than being constrained to `1` through `modulus - 1`), which is itself fine.
2. Line 277 similarly samples a random `r` for use in proving, where a `0` could have detrimental downstream implications due to several multiplication operations. With respect to the IRTF CFRG draft VRF specification:
 - [Section 5.4.2.1 notes](#) that this value `r` should be $1 < r < N-1$.
 - [Section 5.4.2.2 notes](#) that this value `r` should be $0 < r < N-1$.
3. Line 300-301 ensures that the bit-length of `c` is less than modulus bit-length, not that `c` is less than the modulus itself, via rejection sampling. A similar approach is taken for the `s` value on line 311. This approach has trivial implications for the current context, but may become more significant for larger field prime and curve order combinations.
 - The range of the legal values is slightly truncated, introducing bias.
 - The iteration will expose timing information.
4. The implementation of the `proof_to_hash()` function is significantly different from the VRF spec, as it is essentially a combination of the ECVRF Proof to Hash and ECVRF Verifying functions (from specification sections 5.1 and 5.3).

Recommendation

1. Consider rejecting `r == 0` by adding a test and a `continue` statement.
2. If bias is acceptable/intended, add a code comment that includes expected iteration example.

Location

ginger-lib/primitives/src/vrf/ecvrf/mod.rs

Retest Results

2023-03-06 – Fixed

In [pull request 194](#), the Horizen Labs team added a conditional statement testing the value of the random scalar, preventing the generation of the zero value.

19. <https://github.com/HorizenOfficial/ginger-lib/blob/25384653cc9818bbecea1108ef31e5f3930758cf/primitives/src/vrf/ecvrf/mod.rs>



The team also added ample code comments detailing the rejection sampling procedure, the potential security loss, and warning about the non constant-timeness of the implementation.

As a result, this finding is considered *fixed*.



Weak Error Handling via `unwrap()`

Overall Risk Informational
Impact Low
Exploitability Undetermined

Finding ID NCC-E005939-TR4
Component Systemic
Category Error Reporting
Status Fixed

Impact

The use of `unwrap()` within a function returning a `Result` means the caller cannot avoid a panic and an opportunity to deliver an informative error indicator/message is missed. In the (very) extreme, an uncontrolled panic may ultimately become a denial-of-service vector.

Description

The Merkle-related code has several instances of `unwrap()` utilized within functions that return a `Result`. An example from the `append_only/mod.rs` source file is excerpted below.

```
281 fn finalize(&self) -> Result<Self, Error> {
282     let mut copy = (*self).clone();
283
284     if !self.finalized {
285         copy.new_elem_pos[0] = copy.final_pos[0];
286         copy.compute_subtree()?;
287         copy.finalized = true;
288         if copy.array_nodes.len() == 0 {
289             Err(MerkleTreeError::Other("Merkle tree is empty".to_owned()))?
290         }
291         copy.root = *copy.array_nodes.last().unwrap();
292     }
293
294     Ok(copy)
295 }
```

As can be seen above, the enclosing function returns a `Result`, there is a (different) internal check that returns an error, but the final highlighted check via `unwrap()` will cause an uncontrolled panic. Similar instances are within `naive/mod.rs` and `big_lazy_merkle_tree.rs`.

This appears to be systemic across the in-scope repositories, including `zendoo-sc-cryptolib`. Further, a large number of `assert` statements having a similar effect to `unwrap()` appear to go beyond testing critical invariants.

Recommendation

Avoid the `unwrap()` function and `assert` statements in non-test code unless it detects an intentionally unrecoverable error condition and/or a `Result` is not easily utilized. The use of `unwrap()` on configuration constants is arguably fine.

Location

- [ginger-lib/primitives/src/merkle_tree/field_based_mht/append_only/mod.rs](#)
- [ginger-lib/primitives/src/merkle_tree/field_based_mht/naive/mod.rs](#)
- [ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/big_lazy_merkle_tree.rs](#)
- [zendoo-sc-cryptolib/api/src/lib.rs](#)



Retest Results

2023-03-06 – Fixed

In response to this finding, the Horizen Labs team introduced changes in two pull requests, which address the issues outlined in this finding. Specifically,

- [pull request 192](#) for *ginger-lib* introduced specific error types, replaced a number of `unwrap s` in Merkle tree-related code with safer error handling, and added supporting comments when `unwrap s` were safe; and
- [pull request 114](#) for *zendoo-sc-cryptolib* improved error handling by replacing some `assert` and `unwrap` operations with safer logic whenever possible.

These changes are in line with the recommendation above, and this finding is considered *fixed* as a result. However, the project team should strive to follow secure programming practices during the future development of *ginger-lib* and *zendoo-sc-cryptolib* and aim to limit the use of the dangerous patterns described in this finding.



Missing Memory Zeroization on Secret Keys

Overall Risk Informational

Impact Medium

Exploitability Low

Finding ID NCC-E005939-YWJ

Component zendoo-sc-cryptolib

Category Data Exposure

Status Risk Accepted

Impact

If regions of memory become accessible to an attacker, perhaps via a core dump, attached debugger, or disk swapping, the attacker may be able to extract non-cleared secret values.

Description

Typically, all of a function's local stack variables and heap allocations remain in process's memory after the function goes out of scope, unless they are overwritten by new data. This stale data is vulnerable to disclosure through means such as core dumps, an attached debugger, and disk swapping. As a result, sensitive data should be cleared from memory once it goes out of scope.

The different repositories in scope do not exhibit particular care for memory zeroization; in no instance were they observed to erase sensitive data. Specifically, the `Java_com_horizen_schnorrnative_SchnorrSecretKey_nativeFreeSecretKey()` and `Java_com_horizen_vrfnative_VRFSecretKey_nativeFreeSecretKey()` functions implemented in `api/src/lib.rs` do not clear the Schnorr and VRF secret keys, respectively. Additionally, the `SecretKey` type utilized in the `field_based_schnorr.rs` source file does not zeroize memory.

Since the results of memory-clearing functions are not used for functional purposes elsewhere, these functions can become the victim of compiler optimizations and be eliminated. There are a variety of "tricks"²⁰ to attempt to avoid compiler optimizations and ensure that a clearing routine is performed reliably. The Rust community has largely adopted the approach provided by the Zeroize²¹ crate.

Recommendation

While this may be on the margins of the intended use cases, consider utilizing the Zeroize crate to derive the zeroize-on-drop trait for all sensitive values. This may require changes in the dependencies. Ensure the same approach is taken to attach the zeroize-on-drop trait to all secret material found in the Rust bindings. This informational-only finding is documented out of an abundance of caution.

Location

- `zendoo-sc-cryptolib/api/src/lib.rs:L339,L437`
- `ginger-lib/primitives/src/signature/schnorr/field_based_schnorr.rs`

Retest Results

2023-03-06 – Not Fixed

Due to the specific conditions necessary to perform this attack coupled with the complexity to address this finding, the Horizen Labs team decided to postpone the implementation of a fix. An issue was opened on the *ginger-lib* GitHub repository (see [issue #44](#)) to keep track of it.

20. https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_zhao_yang.pdf

21. <https://docs.rs/zeroize/1.5.7/zeroize/>



The status of this finding was updated to *Risk Accepted* as a result.

Client Response

Considering that it doesn't affect our use cases directly, that in order to perform such an attack you'd need to have direct access to the user's machine, and that implementing this in ginger it's quite a big effort, we can postpone this.



6 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.
Low	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
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



7 Non-Security Impacting Observations

This informational 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. Addressing the comments presented in this section would also help improve code quality and consistency.

After retesting, NCC Group noted that the Horizen Labs team diligently addressed a number of the observations below, in [pull request 197](#) for *ginger-lib* and [pull request 118](#) for *zendoo-sc-cryptolib*.

Methodology Notes

The primary testing methodology involved static code review. The consultant team began with a bottom-up review of the implementation within a line-by-line context. This was subsequently complemented by a top-down effort involving planned use cases, documented functionality, system requirements, and desired assurances in the larger context. The two approaches overlap to achieve robust coverage. The consultants also leveraged the different test suites to try and identify shortcomings, to understand the system in more depth and to validate hypotheses.

On Threshold Signature Verification

This section discusses some concerns in the threshold signature verification implementation, as well as potential discrepancies with the *circuit_design.txt* document, which is also provided in the appendix [Circuit Design](#), for reference.

Repeated Iterations during Signature Verification

The threshold signature verification in [zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/constraints/mod.rs](#) is performed via two consecutive iterations; the first one stores the results of the signature verifications in the `verdicts` array, while the second one counts the number of successful verifications from that array, and stores that value in the variable `valid_signatures`, see below. It is unclear why these two iterations could not be collated into one.

```
// Check signatures validity
let mut verdicts = Vec::with_capacity(max_pks);

//Check signatures verification verdict on message
for (i, (pk_g, sig_g)) in validators_keys_updates_g
    .signing_keys_g
    .iter()
    .zip(sigs_g.iter())
    .enumerate()
{
    let v = SchnorrVrfySigGadget::enforce_signature_verdict(
        cs.ns(|| format!("check_sig_verdict_{}", i)),
        pk_g,
        sig_g,
        msg_hash_g.clone(),
    );
    verdicts.push(v);
}

//Count valid signatures
let mut valid_signatures =
    FieldElementGadget::zero(cs.ns(|| "alloc valid signatures count"));
for (i, v) in verdicts.iter().enumerate() {
    valid_signatures = valid_signatures.conditionally_add_constant(
        cs.ns(|| format!("add_verdict_{}", i)),
    );
}
```



```

        v,
        FieldElement::one\(\),
    )?;
}

```

Threshold Considerations

In the circuit design document, the threshold is set as `VALIDATORS_SIZE/2` (see below), while the implementation uses an explicit upper bound distinct from the number of validators.

```
140 require(sigNum > VALIDATORS_SIZE/2)
```

Failure on Invalid Signature

The reference document for the proof verification explicitly fails as soon as a single invalid signature is encountered, see below.

```

// 3 Verify wcert signatures
for (i <- 0 to (VALIDATORS_SIZE-1)) {
    var sigNum = 0 // we will count number of collected signatures
    if (wcert_signatures[i] != NULL) {
        val is_valid_sig = verify_signature(sc_wcert_hash, wcert_signatures[i],
        ↳ signing_keys[i]) // order of signatures should corresponded the order of keys
        ↳ in signing_keys
        require(is_valid_sig == true)
        sigNum = sigNum + 1
    }
}
}

```

In comparison, the implementation counts the number of valid signatures before checking if that number is what it expected; it does not fail immediately upon individual signature verification failure.

Discrepancy in Updated Key Signature Verification

The circuit design specifies that updates to the signing or master keys are signed, by both the old signing and master keys. In the circuit reference document, the *message to be signed* is the key itself, see for example line 149 of that document:

```
val msg_to_sign = updated_signing_keys[i];
```

In the circuit implementation, the code signs and verifies the *hash* of the key instead of the key.

Genesis Constant Always Checked in Implementation

In [zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/constraints/mod.rs](#), the genesis constant (consisting of the hash of the validator tree root concatenated with the threshold value) is checked for every proof verification, while the reference circuit design only checks it right after genesis.

```

// ***** Check genesis constant *****

// Expose genesis_constant as public input
let expected_genesis_constant_g =
    FieldElementGadget::alloc\_input(cs.ns(|| "alloc constant as input"), || {
        Ok(self.genesis_constant)
    })?;

// Check genesis constant
let genesis_constant_g = FieldHashGadget::enforce\_hash\_constant\_length(

```



```

cs.ns(|| "H(MR(genesis_pks), threshold)",
&[
    genesis_validator_keys_tree_root_g.clone(),
    threshold_g.clone(),
],
)?;

genesis_constant_g.enforce_equal(
    cs.ns(|| "genesis_constant: expected == actual"),
    &expected_genesis_constant_g,
)?;

```

Incorrect Capacity

In the function `enforce_validators_key_root()` in `zendoo-sc-cryptolib/demo-circuit/src/native_threshold_sig_w_key_rotation/constraints/data_structures.rs`, the capacity of the vector `validator_mktree_leaves_g` is set to `max_pks`; it should be `2*max_pks` instead since two keys are appended per iteration.

```

fn enforce_validators_key_root<CS: ConstraintSystemAbstract<FieldElement>>(
    mut cs: CS,
    max_pks: usize,
    sig_keys_g: &[SchnorrPkGadget],
    master_keys_g: &[SchnorrPkGadget],
) -> Result<(FieldElementGadget, Vec<FieldElementGadget>), SynthesisError> {
    let height = ((max_pks.next_power_of_two() * 2) as f64).log2() as usize;
    let null_leaf_g: FieldElementGadget = ConstantGadget::from_value(
        cs.ns(|| "hardcoded NULL_LEAF"),
        &GingerMHTParams::ZERO_NODE_CST.unwrap().nodes[0],
    );

    let mut validator_mktree_leaves_g = Vec::with_capacity(max_pks);

    for (i, (signing_key_g, master_key_g)) in
        sig_keys_g.iter().zip(master_keys_g).enumerate() {
        // Enforce signing key hash
        // <snip>

        // Add it to the tree
        validator_mktree_leaves_g.push(signing_key_fe_hash_g);

        // Enforce master key hash
        // <snip>

        // Add it to the tree
        validator_mktree_leaves_g.push(master_key_fe_hash_g);
    }
}

```

Inconsistent Documentation for NULL Signatures

The [documentation](#) describes the null signatures used as placeholder signatures as $\sigma_{NULL} = (0,0)$. However, the implementation defines them as $(1,1)$:

```

21 let e = FieldElement::one();
22 let s = e;
23 let null_sig = SchnorrSig::new(e, s);

```

While both are valid choices for placeholder signatures, the documentation (or implementation) should be updated for consistency.



Confusing Endianness in Field Element Representations

The `zendoo-sc-cryptolib` repository and the few libraries it relies upon use different endianness representations for field elements. The documentation and code comments on that topic are sparse, and the code sometimes follows inconsistent procedures and naming conventions. While no security issue has been *directly* identified as a result of these discrepancies, the NCC Group consultants highlighted a few inconsistencies that could be revised or where additional documentation describing explicit assumptions would be valuable.

On Endianness

As a succinct refresher on endianness and on the representation of integers as byte arrays, let us consider how to store the decimal integer `1122867`, which, in hexadecimal, corresponds to the 32-bit unsigned integer `0x00112233`. The most significant byte of that value is `0x00`, while the least significant byte is `0x33`. In memory, this integer can primarily be stored in one of the following:

- **Big-endian** byte order; where the system stores the *most significant byte* of a word at the *smallest memory address* and the least significant byte at the largest memory address. Using inaccurate addressing between `[0]` and `[3]` for illustrative purposes, that byte array could be stored as follows:

```
[0] 0x00
[1] 0x11
[2] 0x22
[3] 0x33
```

Assuming this quantity is stored as a byte array using C syntax, this array would be displayed by iterating over the indices of the array in increasing order as `{0x00, 0x11, 0x22, 0x33}`.

- **Little-endian** byte order; where the system stores the *least significant byte* at the smallest address.

```
[0] 0x33
[1] 0x22
[2] 0x11
[3] 0x00
```

Similarly, when displayed by iterating over the increasing indices of the array, it would be displayed as `{0x33, 0x22, 0x11, 0x00}`.

The topic of endianness is most often discussed in terms of *bytes*, since it is at that granularity level that memory and I/O operations are performed; bit-level endianness is seldom used. Bit-endianness follows the same principles as outlined above, except that the order is reversed at a *bit-level*: the value used above as example (`1122867`, whose binary representation is `0b100010010001000110011`), when converted to its little-endian representation at a *bit-level* and then displayed in hexadecimal, would be `0xCC448800`.

The NCC Group team noted at least 3 different endianness representations of field elements.

1. When serializing `FieldElements` using the function `serialize_to_buffer()`, *little-endian* byte representation is used. This is also consistent with the function `read_field_element_from_buffer_with_padding()` located in `demo-circuit/src/lib.rs`,



which *right*-pads the buffer with 0s in case the underlying element is shorter than the field order's size.

```
for _ in buff_len..FIELD_SIZE {  
    new_buffer.push(0u8)  
} //Add padding zeros to reach field size
```

2. When serializing `FieldElements` using the function `write_bits()`, *big-endian bit* representation is used.
3. When converting elements to be used by the underlying circuit, *little-endian bit* representation is used, which is *not equivalent* to converting to little-endian bytes. An example of this can be seen in the code excerpt below from the functions `parse_utxo_prover_data()` and `parse_ft_prover_data()` in `api/src/lib.rs`, where the function `write_bits()` is first invoked, before calling the `reverse()` function on that big-endian bit-represented value.

```
// Convert it to bits and reverse them (circuit expects them in LE but write_bits outputs  
↳ in BE)  
let mut sk_bits = sk.write_bits();  
sk_bits.reverse();  
sk_bits.try_into().unwrap()
```

Even though the code base uses these three representations fairly liberally, it seems that these usages are consistent with each other. However, if an element were first converted to a *big-endian bit* representation, reversed, and then interpreted as a *little-endian byte* array, errors would undoubtedly be triggered.

Lone Little-endian Encoding in File `fp.rs`

The file `r1cs/gadgets/std/src/fields/fp.rs` defines various serialization functions to encode a field element into different representations, including bits or bytes. Generally, this file takes the approach of serializing elements into their big-endian representation, as documented in the `to_bits()` function and provided below, for reference.

```
/// Outputs the binary representation of the value in `self` in *big-endian*  
/// form.  
fn to_bits<CS: ConstraintSystemAbstract<F>>(<  
    &self,  
    mut cs: CS,  
) -> Result<Vec<Boolean>, SynthesisError> {  
    self.to_bits_with_length_restriction(&mut cs, 0)  
}
```

However, when serializing field elements to *bytes*, using the function `to_bytes_with_length_restriction()` in `r1cs/gadgets/std/src/fields/fp.rs`, field elements are serialized in little-endian order, as also evidenced by the highlighted loop in the code excerpt below, which trims big-endian bytes first.

```
#[inline]  
pub fn to_bytes_with_length_restriction<CS: ConstraintSystemAbstract<F>>(<  
    &self,  
    mut cs: CS,  
    to_skip: usize,  
) -> Result<Vec<UInt8>, SynthesisError> {  
    let mut byte_values = match self.value {  
        Some(value) => to_bytes!(&value.into_repr())?  
            .into_iter()
```




```

        .map(Some)
        .collect::<Vec<_>>(),
    None => {
        let default = F::default();
        let default_len = to_bytes!(&default).unwrap().len();
        vec![None; default_len]
    }
};

for _ in 0..to_skip {
    byte_values.pop();
}

```

This function is used by both the `to_bytes()` and the `to_bytes_strict()` functions in the same file. Hence, calling either of these two functions will result in a serialized byte-array encoded in little-endian (which is *not* documented in the file), while calling any other serializing function in the same file leads to a result encoded in big-endian. At the very least, consider adding supporting documentation in the form of code comments to detail this discrepancy.

Misleading Endianness Description

As also highlighted above, the implementation sometimes uses slightly misleading terminology when talking about endianness. One example of that can be found in the file [ginger-lib/r1cs/gadgets/std/src/bits/uint8.rs](#), where the functions `into_bits_be()` and `into_bits_le()` mention *byte endianness* (as can be seen in the highlighted comments below), when in reality they refer to bit-endianness. Note that in *big-endian* representation, the orders of *bits* and *bytes* are presumably equivalent, but not in little-endian. This comment also applies to the function `from_bits_le()` in the same file.

```

/// Turns this `UInt8` into its big-endian byte order representation.
pub fn into_bits_be(&self) -> Vec<Boolean> {
    self.bits.iter().rev().cloned().collect()
}

/// Turns this `UInt8` into its little-endian byte order representation.
/// LSB-first means that we can easily get the corresponding field element
/// via double and add.
pub fn into_bits_le(&self) -> Vec<Boolean> {
    self.bits.to_vec()
}

```

On Merkle Tree Primitives

Max Height Validation

The `init()` functions within [ginger-lib/primitives/src/merkle_tree/field_based_mht/append_only/mod.rs](#) and [ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/big_lazy_merkle_tree.rs](#) both validate the height parameter against a maximum size. The `new()` function within [naive/mod.rs](#) does not validate the height parameter. While this Merkle tree has a different use case where the missing validation does not lead to an easily exploitable panic, it may be beneficial to enforce a maximum for consistency or as a simple guard against user error.

R1CS Gadgets Implementation Notes

- In [ginger-lib/r1cs/gadgets/std/src/eq.rs](#), the `conditional_enforce_equal()` and `conditional_enforce_not_equal()` function implementations for slices fail to check



that the slice is non-empty, while the `is_eq()` function correctly performs this check, as highlighted in the code excerpt below.

```
fn is_eq<CS: ConstraintSystemAbstract<ConstraintF>>(&self,
    mut cs: CS,
    other: &Self,
) -> Result<Boolean, SynthesisError> {
    assert_eq!(self.len(), other.len());
    assert!(!self.is_empty());
    // <snip>
}

fn conditional_enforce_equal<CS: ConstraintSystemAbstract<ConstraintF>>(&self,
    mut cs: CS,
    other: &Self,
    condition: &Boolean,
) -> Result<(), SynthesisError> {
    assert_eq!(self.len(), other.len());
    for (i, (a, b)) in self.iter().zip(other).enumerate() {
        // <snip>
    }
    Ok(())
}
```

Poseidon Parameters

The Poseidon hash function²² is based on a sponge construction, in which the internal permutation is composed of successive calls to the round function. Each round function of the Poseidon permutation consists of three layers, 1) AddRoundConstants, 2) SubWords, and 3) MixLayer. While the first and third functions are the same in each round, the number of S-boxes in the second phase differs; the first R_f rounds and last R_f rounds have full S-box layers, while the R_p intermediate rounds have only partial S-box layers. The variables depend on the desired security, rate, and capacity of the instantiation of Poseidon.

There exists a small discrepancy between the reference paper, the script to generate custom parameters for specific curves (developed by the authors of the Poseidon proposal), and the concrete implementation of the Poseidon hash function using the Tweedle curves in the Horizen codebase. Specifically, the reference paper, in Table 2 on page 8, specifies that the variable R_p (i.e., the number of partial S-box rounds) is 57. In contrast, the implementation chooses the value 56, see for example in [ginger-lib/primitives/src/crh/poseidon/parameters/tweedle_dee.rs](https://github.com/ginger-lib/ginger-lib/blob/master/primitives/src/crh/poseidon/parameters/tweedle_dee.rs):

```
impl PoseidonParameters for TweedleFrPoseidonParameters {
    const T: usize = 3; // Size of the internal state (in field elements)
    const R_F: i32 = 4; // Half number of full rounds (the R_f in the paper)
    const R_P: i32 = 56; // Number of partial rounds.
```

22. <https://eprint.iacr.org/2019/458.pdf>



The NCC Group team noted that this latter value was actually consistent with the output of the script used to generate parameters for concrete Poseidon instantiations²³, see the transcript below.

```
$ print(calc_final_numbers_fixed(Crypto.Util.number.getPrime(255), 3, 5, 128, True))  
// [8, 56, 80, 20400]
```

The project team confirmed that this value was still larger than the minimum number of rounds necessary to protect against the different attacks listed in Section 5 of the reference paper, even when accounting for the added *arbitrary* security margin discussed in Section 5.4. As such, this discrepancy does not seem to pose any concrete security risk with regards to the security of the hash function itself. However, it has the potential to introduce interoperability issues.

The same inconsistency was observed by the Horizen Labs team. They later confirmed with the authors of the Poseidon hash function that the quantity for the number of partial rounds was adequate and that the script output was correct.

Bit-length-based Range Checks

Several functions in the code base perform lightweight checks to ensure that an element is in a given field, by ensuring their bit-length is not greater than the bit-length of the field order.

One such example can be seen in the function `scalar_bits_to_constant_length()` in the file `ginger-lib/r1cs/gadgets/std/src/groups/mod.rs`, and excerpted below, for reference.

```
if bits.len() > ScalarF::size_in_bits() {  
    Err(SynthesisError::Other(format!(  
        "Input bits size: {}, max allowed size: {}",  
        bits.len(),  
        ScalarF::size_in_bits()  
    )))?  
}
```

This check does not guarantee that the element is in the field, since an element may be encoded with the same number of bits, yet be larger than the field order. The NCC Group team did not find vulnerable instances of this practice, since in most cases this check is followed by additional validation. Nevertheless, consider checking whether any such instance does assume this check to be final, and adding code comments detailing such assumptions.

Other instances of this behavior happen in

- the file `ginger-lib/primitives/src/signature/schnorr/field_based_schnorr.rs` in the functions `read_checked()` and in the function `is_valid()`;
- the file `ginger-lib/primitives/src/vrf/ecvrf/mod.rs` in the functions `read_checked()` and `is_valid()`;
- the file `ginger-lib/r1cs/gadgets/crypto/src/signature/schnorr/field_based_schnorr.rs` in the function `enforce_signature_computation()`;
- the file `ginger-lib/r1cs/gadgets/crypto/src/vrf/ecvrf/mod.rs` in the function `enforce_proof_to_hash_verification()`;
- the file `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_jacobian.rs` in the function `mul_bits_fixed_base()`;

23. https://extgit.iaik.tugraz.at/krypto/hadhash/-/blob/master/code/calc_round_numbers.py



- the file `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_projective.rs` in the function `mul_bits_fixed_base()`; and
- the file `ginger-lib/r1cs/gadgets/std/src/groups/nonnative/short_weierstrass/short_weierstrass_jacobian.rs` in the function `mul_bits_fixed_base()`.

Elliptic Curve Point Representation and Coordinate Systems

- As also briefly presented in [finding "Problematic Point at Infinity Representation and Comparison"](#) and [finding "Incomplete Elliptic Curve Formulas"](#) the two files `short_weierstrass_projective.rs` and `short_weierstrass_jacobian.rs` are identical (except for trivialities) and both actually implement operations based on *affine* coordinates, but not in *Jacobian* or *projective* coordinates as their respective names seem to indicate.
- There appears to be a small inconsistency in the conventions used to encode curve points to bits by the functions `to_bits()` and `to_compressed()`; the former encodes points as `x || y || inf`, while the latter encodes them as `x || inf || sign(y)`. Note how the infinity flag is once located *before* the y-coordinate, and once *after* the y-coordinate.
- The function `alloc_checked()` also suffers from a slight behavior inconsistency; this function ensures the given point is in the correct subgroup, and does so in one of two ways, depending on the Hamming weight of the cofactor, in order to optimize the number of multiplications:
 - If the Hamming weight of the cofactor is large (larger than the Hamming weight of the subgroup order, r), it simply performs a scalar multiplication of the curve point with $(r - 1)$ and checks that it is equal to $-P$.
 - If the Hamming weight of the cofactor is smaller than the Hamming weight of r , the function first multiplies the point by the inverse of the cofactor (modulo the subgroup order), and then multiplies this result by the cofactor to ensure that the point is in the subgroup of interest.

From an API standpoint, the fact that the first case introduces a constraint (by calling `enforce_equal()`) while the other case “fixes” the point could be misleading.

Additionally note that the multiplication by the inverse of the cofactor works only in case it admits an inverse, namely if $\text{GCD}(\text{cofactor}, r) = 1$.

Finally, an incorrect comment is present on [line 1193](#) of that file. The operation performed is a multiplication by $r - 1$ and not by $r - 2$, as stated.

```
// If we multiply by r, we actually multiply by r - 2.
let r_minus_1 = (-P::ScalarField::one()).into_repr();
```

Bowe-Hopwood Hash

- In the code snippet below, excerpted from the function `evaluate()` in the file `ginger-lib/primitives/src/crh/bowe_hopwood/mod.rs`, the capacity of the `padded_input` vector should be set to `8*input.len()`, since the input gets converted to bits before being copied to `padded_input`.
- It is also unclear why this function checks whether the padding step failed (second highlighted portion below), since it seems very unlikely that this would happen.

```
let mut padded_input = Vec::with_capacity(input.len());
let input = bytes_to_bits(input);
// Pad the input if it is not the current length.
padded_input.extend_from_slice(&input);
if input.len() % CHUNK_SIZE != 0 {
    let current_length = input.len();
```



```

        for _ in 0..(CHUNK_SIZE - current_length % CHUNK_SIZE) {
            padded_input.push(false);
        }
    }

    if padded_input.len() % CHUNK_SIZE != 0 {
        Err(Box::new(CryptoError::Other(
            format!(
                "Input is not multiple of the chunk size. Input len: {}, chunk size: {}",
                padded_input.len(),
                CHUNK_SIZE,
            )
        ).to_owned(),
    )))?
    }
}

```

Schnorr Signatures

- In the file [ginger-lib/primitives/src/signature/schnorr/field_based_schnorr.rs](#), if the function `to_field_elements()` ever returned the representation of a point using a coordinate system composed of more than two elements (such as projective coordinates which represent a point using a triple of elements), a number of issues could occur, one of them being the hardcoded initialization of the hash function with a 4-element capacity.

```

//Affine coordinates of R (even if R is infinity)
let r_coords = r.to_field_elements()?;

// Compute e = H(m || R || pk.x)
let e = {
    let mut digest = H::init_constant_length(4, None);
    digest.update(message);
    r_coords.into_iter().for_each(|coord| {
        digest.update(coord);
    });
    digest.update(pk_coords[0]);
    digest.finalize()
};

```

- Additionally, public keys are not checked to be valid prior to signature verification, both in this file and in the function `enforce_signature_computation()` in the file [ginger-lib/r1cs/gadgets/crypto/src/signature/schnorr/field_based_schnorr.rs](#). While this is somewhat documented, for example in the description of `FieldBasedSchnorrSignatures` `cheme`, see below, it would be advisable to either add public key validation, or more thoroughly document the lack of validation in both signature verification functions.

```

// Low-level crypto for the length-restricted Schnorr Signature, does not perform any
// input validity check. It's responsibility of the caller to do so, through keyverify()
// function for the PublicKey, read() or is_valid() functions for
↳ FieldBasedSchnorrSignature.

```

Constraint System Primitives

- The `addmany()` function implemented for `UInt32` (in [ginger-lib/r1cs/gadgets/std/src/bits/uint32.rs](#)) and for `UInt64` (in [ginger-lib/r1cs/gadgets/std/src/bits/uint64.rs](#)) has a



slight discrepancy in input validation logic. Namely, the implementation for `UInt64` allows a single operand, see below,

```
assert!(operands.is_empty());
assert!(operands.len() <= 10);

if operands.len() == 1 {
    return Ok(operands[0].clone());
}
```

whereas the implementation for `UInt32` fails if the number of operands is less than two.

```
assert!(operands.len() >= 2); // Weird trivial cases that should never happen
                                // TODO: Check this bound. Is it really needed ?
assert!(operands.len() <= 10);
```

Additionally, the arbitrary upper bound could either be removed, or documented.

- In the function `scalar_bits_to_constant_length()` in `ginger-lib/r1cs/gadgets/std/src/groups/mod.rs`, some operations are currently commented out. Consider deleting them.

```
// bits per limb must not exceed the CAPACITY minus one bit, which is
// reserved for the addition.
let bits_per_limb = std::cmp::min(
    (ConstraintF::Params::CAPACITY - 1) as usize,
    ScalarF::size_in_bits(),
);
// ceil(ScalarF::size_in_bits())/bits_per_limb
// let num_limbs = (ScalarF::size_in_bits() + bits_per_limb - 1)/bits_per_limb;
```

- In the file `ginger-lib/algebra/src/biginteger/mod.rs`, the trait definition of `BigInteger` has documentation stating that the `to_bits()` function returns a representation without leading zeros, which seems incorrect in practice, as also noted by the `TODO` that follows.

```
/// Returns the bit representation in a big endian boolean array, without
/// leading zeros.
// TODO: the current implementation does not seem to skip leading zeroes.
// Let us check its usage and determine if a change is reasonable.
fn to_bits(&self) -> Vec<bool>;
```



8 Circuit Design

Main features of v1.3 version:

1. Every validator has two keys: signing key and master key:
 - the signing key is a hot key that is used to sign withdrawal certificates;
 - the master key is a cold key that is kept offline and used only to change the signing key or the master key itself.
2. To allow rotation of keys every certificate contains an additional custom field which
 - ↳ contains a root hash of a Merkle tree with the keys of all validators for the next epoch.
3. Each validator can change his signing key by submitting a request that is signed by his master and old signing key.
 - ↳ master and old signing key.
 - Then, the new key should substitute the old one in the Merkle tree of keys committed in the withdrawal certificate.
 - ↳ the withdrawal certificate.
 - The SNARK proof should verify both the signed request and the correct update of the tree.

The same way a validator can change his master key (e.g., in case he wants to transfer his validator rights to someone else).

↳ validator rights to someone else).

The request with the new master key should be signed by the old master key and signing key.
4. The withdrawal certificate should be signed by a majority of signing keys committed in the previous WCert (or genesis_constant if it is the first WCert)
 - ↳ previous WCert (or genesis_constant if it is the first WCert)

```
=====
↳ =====
1. Withdrawal Certificate Structure:
=====
↳ =====
```

```
// WCert structure
type WithdrawalCertificate {
    ledgerId: Int,                // unique ledger id defined by the MC
    epochId: Int,
    bt_list: List[BackwardTransfer],
    quality: Int,
    mcb_sc_txs_com: Hash,         // the cumulative SCTxsCommitment hash taken
    ↳ from the last MC block in the proved epoch).
    ft_min_fee,                  // minimal fee for a forger transfer (used
    ↳ by MC to filter FTs in the epoch after the next)
    btr_min_fee,                 // minimal fee for a backward transfer
    ↳ request

    proof_data: {
        scb_new_mst_root: Hash,   // sidechain state
        scb_validators_keys_root: Hash, // Merkle root hash of validators keys for
        ↳ the next epoch
    },
    proof: POV_MC_WCert
}

// WithdrawalCertificateData is a structure composed from a WithdrawalCertificate, it is
↳ basically re-structured WithdrawalCertificate (without proof)
// used to efficiently pass certificate's data everywhere in the proofs. It also defines
↳ how a withdrawal certificate is hashed on the sidechain side.
// Note that this is also how the MC hashes a certificate when constructing
↳ SCTxsCommitment and public input for POV_MC_WCert.
```



```
// wcert_data_hash = H(sys_data | proof_data), where 'sys_data' and 'proof_data' are
↳ hashes of their elements concatenation.
type WithdrawalCertificateData {
  sys_data: {
    ledgerId: Int,
    epochId: Int,
    bt_list_hash: MERKLE_ROOT,           // Merkle root hash of all BTs from the
    ↳ certificate (recall that MC hashes all complex proof_data params from the
    ↳ certificate)
    quality: Int,
    mcb_sc_txs_com: Hash
    ft_min_fee,
    btr_min_fee,
  }
  proof_data: {
    scb_new_mst_root: Hash,
    scb_validators_keys_root: Hash,
  }
}
```

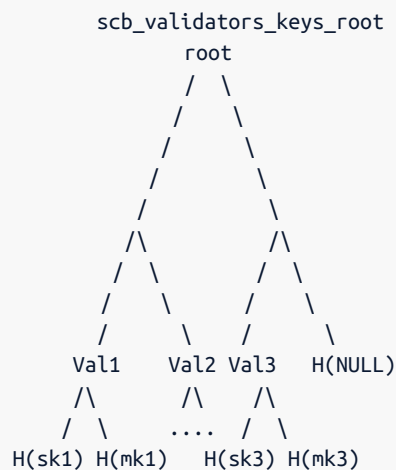
↳ =====

2. Validator Keys Merkle Tree

↳ =====

↳ =====

Example of a tree for the sidechain with 3 validators (the number of validators can be
↳ different)



where

Val1 - node that represents hash of keys of validator 1
H(sk1) - hash of the signing key of validator 1
H(mk1) - hash of the master key of validator 1
the same for other validators

↳ =====

3. SNARK proof for a withdrawal certificate:

↳ =====

```
POV_MC_WCert {
  Setup keys: {
```




```

    pk = pk_mc_wcert
    vk = vk_mc_wcert
}
Input: {
    genesis_constant,           // Passed directly by the MC. Contains the Merkle
    ↳ root hash of the tree with INITIAL set of validator keys
    sc_prev_wcert_hash,         // Passed directly by the MC. Note: it is assumed
    ↳ that sc_prev_wcert_hash == NULL when the very first WCert is proven (depends on
    ↳ implementation)
    sc_wcert_hash: Hash,         // hash of the proven withdrawal certificate as
    ↳ defined by WithdrawalCertificateData.
                                // Computed directly by the MC from
                                ↳ WithdrawalCertificateData
}
Witness: {
    wcert,                      // currently proven wcert
    prev_wcert,                 // previous withdrawal certificate
    wcert_signatures: Vector[VALIDATORS_SIZE], // signatures of validators, ordered
    ↳ in accordance with keys in the signing_keys vector.
                                // Some elements may contain NULLs which means there is
                                ↳ no sigs of these validators.
    signing_keys: Vector[VALIDATORS_SIZE], // signing keys of validators
    ↳ committed in the previous WCert in the format: (sk1, sk2, sk3,...)
    master_keys: Vector[VALIDATORS_SIZE], // master keys of validators
    ↳ committed in the previous WCert in the format: (mk1, mk2, mk3, ...)
    updated_signing_keys: Vector[VALIDATORS_SIZE], // a new set of signing keys
    ↳ for the next epoch
    updated_master_keys: Vector[VALIDATORS_SIZE], // a new set of master keys
    ↳ for the next epoch

    // if there is an update of any key from 'signing_keys' or 'master_keys', the new
    ↳ key should be signed by both old SIGNING and MASTER keys
    updated_signing_keys_sk_signatures: Vector[VALIDATORS_SIZE], // signatures made
    ↳ with old SIGNING keys for new SIGNING keys, elements can be empty if there was no
    ↳ update
    updated_signing_keys_mk_signatures: Vector[VALIDATORS_SIZE], // signatures made
    ↳ with old MASTER keys for new SIGNING keys, elements can be empty if there was no
    ↳ update
    updated_master_keys_sk_signatures: Vector[VALIDATORS_SIZE], // signatures made
    ↳ with old SIGNING keys for new MASTER keys, elements can be empty if there was no
    ↳ update
    updated_master_keys_mk_signatures: Vector[VALIDATORS_SIZE], // signatures made
    ↳ with old MASTER keys for new MASTER keys, elements can be empty if there was no
    ↳ update
}

statement: {
    // 1 Verify certificates
    if (sc_prev_wcert_hash != NULL)
        require(sc_prev_wcert_hash == H(prev_wcert))
    require(sc_wcert_hash == H(wcert))

    // 2 Verify validators signing keys
    val validators_keys_root = reconstruct_merkle_root_hash(signing_keys|
    ↳ master_keys); // it should build a tree as in picture above from the zipped
    ↳ vector (sk1, mk1, sk2, mk2, ...)
    if (sc_prev_wcert_hash != NULL)
        require(prev_wcert.proof_data.scb_validators_keys_root == validators_keys_root)
}

```



```

else
    require(genesis_constant == validators_keys_root)

// 3 Verify wcert signatures
for (i <- 0 to (VALIDATORS_SIZE-1)) {
    var sigNum = 0 // we will count number of collected signatures
    if (wcert_signatures[i] != NULL) {
        val is_valid_sig = verify_signature(sc_wcert_hash, wcert_signatures[i],
        ↳ signing_keys[i]) // order of signatures should corresponded the order
        ↳ of keys in signing_keys
        require(is_valid_sig == true)
        sigNum = sigNum + 1
    }
}
require(sigNum > VALIDATORS_SIZE/2)

// 4 Verify quality (We assume `quality` is the number of signatures in
↳ `signatures` vector)
require(quality == sigNum)

// 5 Verify signatures of updated validators keys
for (i <- 0 to (VALIDATORS_SIZE-1)) {

    if (updated_signing_keys[i] != signing_keys[i]) {
        val msg_to_sign = updated_signing_keys[i];
        val sk_sig = updated_signing_keys_sk_signatures[i];
        val mk_sig = updated_signing_keys_mk_signatures[i];
        val is_valid_sk_sig = verify_signature(msg_to_sign, sk_sig,
        ↳ signing_keys[i]) // updated signing key should be signed old signing
        ↳ key
        val is_valid_mk_sig = verify_signature(msg_to_sign, mk_sig,
        ↳ master_keys[i]) // updated signing key should be signed old master
        ↳ key
        require(is_valid_sk_sig == is_valid_mk_sig == true)
    }

    if (updated_master_keys[i] != master_keys[i]) {
        val msg_to_sign = updated_master_keys[i];
        val sk_sig = updated_master_keys_sk_signatures[i];
        val mk_sig = updated_master_keys_mk_signatures[i];
        val is_valid_sk_sig = verify_signature(msg_to_sign, sk_sig,
        ↳ signing_keys[i]) // updated master key should be signed old signing
        ↳ key
        val is_valid_mk_sig = verify_signature(msg_to_sign, mk_sig,
        ↳ master_keys[i]) // updated master key should be signed old master key
        require(is_valid_sk_sig == is_valid_mk_sig == true)
    }
}

// 6 Verify new validators keys
val new_validators_keys_root = reconstruct_merkle_root_hash(updated_signing_keys |
↳ updated_master_keys); // it should build a tree as in picture above from the
↳ vector (sk1, mk1, sk2, ...)
require(wcert.proof_data.scb_validators_keys_root == new_validators_keys_root)
}
}

```



9 Appendix: Additional Scoping Detail

The following source code files from the [zendoo-sc-cryptolib repository](#), [commit 2007463](#) provide the first portion of in-scope functionality under review.

1. `zendoo-sc-cryptolib/api/src/cctp_calls.rs`
2. `zendoo-sc-cryptolib/api/src/exception.rs`
3. `zendoo-sc-cryptolib/api/src/lib.rs`
4. `zendoo-sc-cryptolib/api/src/utils.rs`
5. `zendoo-sc-cryptolib/demo-circuit/src/common/mod.rs`
6. `zendoo-sc-cryptolib/demo-circuit/src/common/constraints/mod.rs`
7. `zendoo-sc-cryptolib/demo-circuit/src/common/data_structures.rs`
8. `zendoo-sc-cryptolib/demo-circuit/src/constants/mod.rs`
9. `zendoo-sc-cryptolib/demo-circuit/src/constants/old_circuit_keys.rs`
10. `zendoo-sc-cryptolib/demo-circuit/src/constants/personalizations.rs`
11. `zendoo-sc-cryptolib/demo-circuit/src/lib.rs`
12. `zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/constraints/mod.rs`
13. `zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/constraints/data_structures.rs`
14. `zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/mod.rs`
15. `zendoo-sc-cryptolib/demo-circuit/src/naive_threshold_sig_w_key_rotation/data_structures.rs`
16. `zendoo-sc-cryptolib/demo-circuit/src/type_mapping.rs`
17. `zendoo-sc-cryptolib/demo-circuit/src/utils.rs`
18. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/certnative/BackwardTransfer.java`
19. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/certnative/CreateProofResult.java`
20. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/certnative/NaiveThresholdSignatureWKeyRotation.java`
21. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/certnative/NaiveThresholdSigProof.java`
22. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/certnative/WithdrawalCertificate.java`
23. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/commitmenttreenative/CommitmentTree.java`
24. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/commitmenttreenative/CustomBitvectorElementsConfig.java`
25. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/commitmenttreenative/CustomFieldElementsConfig.java`
26. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/commitmenttreenative/ScAbsenceProof.java`
27. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/commitmenttreenative/ScExistenceProof.java`
28. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/cswnative/CswFtProverData.java`
29. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/cswnative/CswProof.java`
30. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/cswnative/CswSysData.java`
31. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/cswnative/CswUtxoProverData.java`
32. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/fwtnative/ForwardTransferOutput.java`
33. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/librustsidechains/Constants.java`
34. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/librustsidechains/FieldElement.java`
35. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/librustsidechains/Library.java`
36. `zendoo-sc-cryptolib/jni/src/main/java/com/horizen/librustsidechains/Utils.java`



37. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/merkletreenative/InMemoryAppendOnlyMerkleTree.java
38. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/merkletreenative/InMemorySparseMerkleTree.java
39. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/merkletreenative/MerklePath.java
40. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/poseidonnative/PoseidonHashable.java
41. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/poseidonnative/PoseidonHash.java
42. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/provingsystemnative/ProvingSystem.java
43. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/provingsystemnative/ProvingSystemType.java
44. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/schnornnative/SchnorrKeyPair.java
45. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/schnornnative/SchnorrPublicKey.java
46. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/schnornnative/SchnorrSecretKey.java
47. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/schnornnative/SchnorrSignature.java
48. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/schnornnative/ValidatorKeysUpdatesList.java
49. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/scutxonative/ScUtxoOutput.java
50. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/vrfnative/VRFKeyPair.java
51. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/vrfnative/VRFProof.java
52. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/vrfnative/VRFProveResult.java
53. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/vrfnative/VRFPublicKey.java
54. zendoo-sc-cryptolib/jni/src/main/java/com/horizen/vrfnative/VRFSecretKey.java

The following source code files from the [ginger-lib repository, commit 2538465](#) provide the second portion of in-scope functionality under review.

1. ginger-lib/primitives/src/crh/bowe_hopwood/mod.rs
2. ginger-lib/primitives/src/merkle_tree/field_based_mht/append_only/mod.rs
3. ginger-lib/primitives/src/merkle_tree/field_based_mht/mod.rs
4. ginger-lib/primitives/src/merkle_tree/field_based_mht/naive/mod.rs
5. ginger-lib/primitives/src/merkle_tree/field_based_mht/parameters/mod.rs
6. ginger-lib/primitives/src/merkle_tree/field_based_mht/parameters/tweedle_dee.rs
7. ginger-lib/primitives/src/merkle_tree/field_based_mht/parameters/tweedle_dum.rs
8. ginger-lib/primitives/src/merkle_tree/field_based_mht/path.rs
9. ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/big_lazy_merkle_tree.rs
10. ginger-lib/primitives/src/merkle_tree/field_based_mht/smt/mod.rs
11. ginger-lib/primitives/src/signature/mod.rs
12. ginger-lib/primitives/src/signature/schnorr/field_based_schnorr.rs
13. ginger-lib/primitives/src/vrf/ecvrf/mod.rs
14. ginger-lib/primitives/src/vrf/mod.rs
15. ginger-lib/r1cs/core/src/constraint_system.rs
16. ginger-lib/r1cs/core/src/error.rs



17. `ginger-lib/r1cs/core/src/impl_constraint_var.rs`
18. `ginger-lib/r1cs/core/src/impl_lc.rs`
19. `ginger-lib/r1cs/core/src/lib.rs`
20. `ginger-lib/r1cs/gadgets/crypto/src/crh/mod.rs`
21. `ginger-lib/r1cs/gadgets/crypto/src/crh/bowe_hopwood/mod.rs`
22. `ginger-lib/r1cs/gadgets/crypto/src/crh/poseidon/mod.rs`
23. `ginger-lib/r1cs/gadgets/crypto/src/crh/poseidon/tweedle/density_optimized/constants.rs`
24. `ginger-lib/r1cs/gadgets/crypto/src/crh/poseidon/tweedle/density_optimized/mod.rs`
25. `ginger-lib/r1cs/gadgets/crypto/src/crh/poseidon/tweedle/mod.rs`
26. `ginger-lib/r1cs/gadgets/crypto/src/crh/sbox.rs`
27. `ginger-lib/r1cs/gadgets/crypto/src/lib.rs`
28. `ginger-lib/r1cs/gadgets/crypto/src/merkle_tree/field_based_mht/mod.rs`
29. `ginger-lib/r1cs/gadgets/crypto/src/merkle_tree/mod.rs`
30. `ginger-lib/r1cs/gadgets/crypto/src/signature/mod.rs`
31. `ginger-lib/r1cs/gadgets/crypto/src/signature/schnorr/field_based_schnorr.rs`
32. `ginger-lib/r1cs/gadgets/crypto/src/vrf/ecvrf/mod.rs`
33. `ginger-lib/r1cs/gadgets/crypto/src/vrf/mod.rs`
34. `ginger-lib/r1cs/gadgets/std/src/alloc.rs`
35. `ginger-lib/r1cs/gadgets/std/src/bits/boolean.rs`
36. `ginger-lib/r1cs/gadgets/std/src/bits/mod.rs`
37. `ginger-lib/r1cs/gadgets/std/src/bits/uint32.rs`
38. `ginger-lib/r1cs/gadgets/std/src/bits/uint64.rs`
39. `ginger-lib/r1cs/gadgets/std/src/bits/uint8.rs`
40. `ginger-lib/r1cs/gadgets/std/src/eq.rs`
41. `ginger-lib/r1cs/gadgets/std/src/fields/fp.rs`
42. `ginger-lib/r1cs/gadgets/std/src/fields/mod.rs`
43. `ginger-lib/r1cs/gadgets/std/src/groups/curves/mod.rs`
44. `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/mod.rs`
45. `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_jacobian.rs`
46. `ginger-lib/r1cs/gadgets/std/src/groups/curves/short_weierstrass/short_weierstrass_projective.rs`
47. `ginger-lib/r1cs/gadgets/std/src/groups/mod.rs`
48. `ginger-lib/r1cs/gadgets/std/src/instantiated/mod.rs`
49. `ginger-lib/r1cs/gadgets/std/src/instantiated/tweedle/curves.rs`
50. `ginger-lib/r1cs/gadgets/std/src/instantiated/tweedle/fields.rs`
51. `ginger-lib/r1cs/gadgets/std/src/instantiated/tweedle/mod.rs`
52. `ginger-lib/r1cs/gadgets/std/src/lib.rs`
53. `ginger-lib/r1cs/gadgets/std/src/select.rs`
54. `ginger-lib/r1cs/gadgets/std/src/to_field_gadget_vec.rs`

