# Zendoo Proof Verifier Cryptography Review

## Zen Blockchain Foundation

November 29, 2021 – Version 1.2


**Prepared for**

Luca Cermelli
Daniele Di Benedetto
Alberto Garoffolo
Rosario Pabst

**Prepared by**

Paul Bottinelli
Ava Howell
Eli Sohl

# Executive Summary

## Synopsis

During the summer of 2021, Horizen Labs engaged NCC Group to conduct a cryptography review of Zendoo protocol's proof verifier. This system generates and verifies modified Marlin proofs with a polynomial commitment scheme based on the hardness of the discrete logarithm problem in prime-order groups. The system also provides optimized batch verification of accumulated proofs. The review included a large number of supporting elements for the proof system, such as the underlying field arithmetic, instantiations of specific elliptic curves, a custom hash function, and optimized Merkle Tree implementations. NCC Group assigned three consultants for a total of 42 person-days over the course of five calendar weeks on this review.

Following this review, NCC Group performed a retest of the findings uncovered during the initial engagement a few weeks later.

## Scope

NCC Group's evaluation included:

- Selected portions of the `ginger-lib` repository: github.com/HorizenOfficial/ginger-lib on branch `development_tmp` at commit `b8b3a9feb8f1c4dde5ce3a3f2e951d597ec9d696`. More specifically:
  - Field and BigInteger arithmetic and their corresponding serialization and deserialization functions,
  - Assembly optimizations of the underlying arithmetic,
  - Tweedle Curves and their corresponding fields,
  - Multi Scalar Multiplication (MSM) and Fast Fourier Transforms (FFT),
  - Implementation of the snark-friendly hash function Poseidon,
  - Concrete parameter instantiation of the Poseidon hash for the Tweedle curves,
  - Merkle trees and paths implementations,
  - Coboundary Marlin and Final Darlin batch verification and accumulation using Discrete Log Accumulators;
- Marlin implementation: https://github.com/HorizenLabs/marlin on branch `dev` at commit `eaf2a6a4ebfbb8034f158583f29179765a2f5297`;
- Polynomial commitment implementation: https://github.com/HorizenLabs/poly-commit on branch `dev` at commit `7d8a0f38c218229288c8885fb416b4005f9f7d59`, including pull request 28: Proof size optimization;
- `zendoo-cctp-lib` to support cross chain transfers for the Zendoo protocol: https://github.com/HorizenOfficial/zendoo-cctp-lib on branch `dev` at commit `f7aeeba5266a2a6d82e2186958d11ead165191ab`;
- `zendoo-mc-cryptolib`, an FFI library crate that exposes the ginger-lib Rust components needed to support Zendoo in mainchain: https://github.com/HorizenOfficial/zendoo-mc-cryptolib on branch `sync_with_cctp_lib` at commit `ac1a8d59330953d9bfabf8c65b11b21bde6669f9`.

## Limitations

Due to the large size 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.

At the time of the review, some portions of the code were still under development, as evidenced by a number of "TODO"s throughout the repositories and some commented code portions. The NCC Group team also performed the review on dedicated development branches, which eventually will have to be completed and integrated within the larger Zendoo ecosystem.

Additionally, side-channel attacks leveraging timing leaks were not an area of concern for the Horizen Labs team and as such non constant-time operations were not investigated in detail.

Finally, the changes introduced in the different pull requests prior to the retest sometimes contained modifications to files that were out of the initial scope. These updates were not reviewed in great depth.

## Key Findings

The NCC Group team reported a total of 22 findings during the course of the engagement. The most notable findings were:

- **Missing Polynomial Normalization after Arithmetic Operations:** Incorrect polynomial representation resulting from arithmetic operations may break assumptions and lead to erroneous computations or may result in denial of service attacks via Rust panics.

- **Batch Proof Verification Bypass:** A maliciously crafted set of proofs or tampered verification keys may pass the batch (and aggregated) verification procedure. This might allow attackers to tamper with proofs without legitimate users noticing, potentially impacting the trust in the zero-knowledge proof system.
- **Incorrect Random Polynomial Generation:** The generation of masking polynomials with inadequate random coefficients may invalidate the security proofs and breach the zero-knowledge property.
- **Missing Length Check in Canonical Deserialization:** Different serialized field elements may be deserialized to the same value, resulting in potentially adverse and unexpected consequences, including breach of consensus.
- **No Domain Separation in Merkle Tree Implementation:** An attacker may be able to produce a series of leaves which allows them to forge an inclusion proof in the Merkle tree.
- **Merkle Leaf Nodes Not Zeroed on Reset:** Incorrect values may be computed for root nodes, subtree nodes, and tree paths. Computed values may not be reproducible between users or between consecutive program executions.

The NCC Group team also collected a number of informational engagement notes which are provided in Appendix B on page 50.

**After retesting**, NCC Group found that a large majority of the findings had been addressed. Out of a total of twenty-two (22) original findings, fourteen (14) were marked as *Fixed* and one (1) as *Partially Fixed*. Additionally, three (3) findings were marked as *False Positive* and four (4) were marked as *Risk Accepted*, after discussions with the Horizen Labs team.

## Strategic Recommendations

Consider cleaning up the different repositories by deleting all unused code. The current code bases are large, and contain a lot of unused, outdated, or otherwise unnecessary implementations. This makes the code bases more difficult to maintain and eventually increases the attack surface.

In order to provide more assurance regarding the lack of exploitable vulnerabilities (for example, in the presence of adverse input parameters), more comprehensive unit tests could be written, particularly around some of the higher-level primitives such as proof aggregation and verification. Randomized input testing via fuzzing might be a valuable approach to uncover potential additional edge cases. The Rust `cargo fuzz` subcommand is an easy-to-use wrapper around libFuzzer.

Due to the deep function hierarchy, it might not always be evident if and where parameter validation is performed. As such, consider revisiting some of the existing functions to assess whether stricter input validation is necessary. Avoid the use of unsafe Rust code that can cause panics, and catch possible errors with informative error messages where possible.

The code base could also benefit from more specific and detailed comments, given the complex nature of the performed operations. Additionally, ensuring that the reference papers and the implementation use the exact same terminology for variable and function naming would greatly help readers follow the flow of complex cryptographic operations.

# Dashboard

## Target Metadata

| | |
|---|---|
| **Name** | Zendoo Proof Verifier |
| **Type** | Cryptographic Libraries |
| **Platforms** | Rust with C FFI |
| **Environment** | Local |

## Engagement Data

| | |
|---|---|
| **Type** | Cryptography Implementation Review |
| **Method** | Code-assisted |
| **Dates** | 2021-06-07 to 2021-07-09 |
| **Consultants** | 3 |
| **Level of Effort** | 42 person-days |

## Targets

| | |
|---|---|
| `ginger-lib` | A general purpose zk-SNARK library supporting recursive proof composition: https://github.com/HorizenOfficial/ginger-lib |
| `poly-commit` | A Rust library that implements (univariate) polynomial commitment schemes: https://github.com/HorizenLabs/poly-commit |
| `marlin` | A Rust library that implements a preprocessing zkSNARK for R1CS with universal and updatable SRS: https://github.com/HorizenLabs/marlin |
| `zendoo-cctp-lib` | A Rust library supporting Cross Chain Transfers for Zendoo Protocol: https://github.com/HorizenOfficial/zendoo-cctp-lib |
| `zendoo-mc-cryptolib` | An FFI library crate that exposes the ginger-lib Rust components needed to support Zendoo in mainchain: https://github.com/HorizenOfficial/zendoo-mc-cryptolib |

## Finding Breakdown

| | Original Assessment | Remaining |
|---|---|---|
| Critical issues | 0 | 0 |
| High issues | 3 | 0 |
| Medium issues | 3 | 1 |
| Low issues | 10 | 2 |
| Informational issues | 3 | 2 |

## Category Breakdown

| | |
|---|---|
| Cryptography | 2 |
| Data Exposure | 1 |
| Denial of Service | 1 |
| Other | 1 |

## Component Breakdown

| | |
|---|---|
| Systemic | 1 |
| ginger-lib | 4 |

## Key

Critical High Medium Low Informational

# 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. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 48.

| Title | Status | ID | Risk |
|---|---|---|---|
| Missing Polynomial Normalization after Arithmetic Operations | Fixed | 009 | High |
| Batch Proof Verification Bypass | Fixed | 016 | High |
| Incorrect Random Polynomial Generation | Fixed | 017 | High |
| Missing Length Check in Canonical Deserialization | False Positive | 001 | Medium |
| No Domain Separation in Merkle Tree Implementation | Risk Accepted | 010 | Medium |
| Merkle Leaf Nodes Not Zeroed on Reset | Fixed | 015 | Medium |
| Incorrect Hiding Bound in Labeled Polynomial Commitment | Fixed | 022 | Medium |
| Secure Rust Best Practices Not Always Followed | Partially Fixed | 002 | Low |
| Misleading Modular Reduction Function | Fixed | 004 | Low |
| Potential Panic with Zero-Division | Fixed | 005 | Low |
| Outdated and Vulnerable Rust Dependencies | Fixed | 006 | Low |
| Insufficient Parameter Checks in Multi-Scalar Multiplication | Fixed | 008 | Low |
| Insufficient Parameter Validation in Merkle Tree Implementation | Fixed | 011 | Low |
| Potential DoS via Memory Exhaustion in Merkle Tree Instantiation | Risk Accepted | 012 | Low |
| Incoherence in Poseidon Round Number Parameters | False Positive | 013 | Low |
| RNG Implementation Non-Compliant with Rust Documentation | Fixed | 014 | Low |
| Ambiguous Fiat-Shamir Oracle Instantiation and Input Serialization | Fixed | 018 | Low |
| Discrepancy with Reference Paper on Random Challenge Domain | False Positive | 019 | Low |
| Undefined Behavior in Foreign Function Interface | Fixed | 021 | Low |
| Non Constant-Time Modular Exponentiation | Risk Accepted | 003 | Informational |
| Missing Memory Zeroization | Risk Accepted | 007 | Informational |
| Potential to Randomly Generate Trivial Random Challenges | Fixed | 020 | Informational |

| | |
|---|---|
| **Finding** | **Missing Polynomial Normalization after Arithmetic Operations** |
| Risk | **High** Impact: Medium, Exploitability: Medium |
| Identifier | NCC-E001741-009 |
| Status | Fixed |
| Category | Data Validation |
| Component | ginger-lib |
| Location | `algebra/src/fft/polynomial/dense.rs` |
| Impact | Incorrect polynomial representation resulting from arithmetic operations may break assumptions and lead to erroneous computations or may result in denial of service attacks via Rust panics. |

**Description**

The file `fft/polynomial/dense.rs` provides an implementation of *dense* polynomials to be used for FFTs. These polynomials are represented by vectors in which each entry corresponds to a coefficient. These coefficients are elements of a finite field, and as such, the sum of two coefficients may take any value in the range $0, \ldots, p-1$, where $p$ is the order of the prime field.

When adding two polynomials of the same degree using the function `add()`, trailing coefficients that sum to zero are not trimmed. This contradicts an underlying assumption on the shape of polynomial representations, namely that the coefficient of the leading term is non-zero.

As an example, summing the polynomials $3 + 2x + x^2$ and $1 + (p-1)x^2$ (using the function `add()` provided below for reference) represented by the vectors `[3, 2, 1]` and `[1, 0, p – 1]` will result in the vector `[4, 2, 0]`, namely the trailing position is equal to zero.

```
fn add(self, other: &'a DensePolynomial<F>) -> DensePolynomial<F> {
    if self.is_zero() {
        other.clone()
    } else if other.is_zero() {
        self.clone()
    } else {
        if self.degree() >= other.degree() {
            let mut result = self.clone();
            for (a, b) in result.coeffs.iter_mut().zip(&other.coeffs) {
                *a += b
            }
            result
        } else {
            let mut result = other.clone();
            for (a, b) in result.coeffs.iter_mut().zip(&self.coeffs) {
                *a += b
            }
            // If the leading coefficient ends up being zero, pop it off.
            while result.coeffs.last().unwrap().is_zero() {
                result.coeffs.pop();
            }
            result
        }
    }
}
```

Interestingly, note that the `else`-clause in the `add()` function above does perform this trimming.

While this failure to trim leading zero coefficients is technically not inconsistent with the current polynomial representation (and should not lead to incorrect results), the implementation assumes that all trailing zeros have been trimmed from polynomials.

As a result, functions like `degree()` (provided below) will panic on unexpected inputs.

```
/// Returns the degree of the polynomial.
pub fn degree(&self) -> usize {
    if self.is_zero() {
        0
    } else {
        assert!(self.coeffs.last().map_or(false, |coeff| !coeff.is_zero()));
        self.coeffs.len() - 1
    }
}
```

This oversight with regards to the trimming of zero coefficients applies to function `add_assign()`, `sub()` and `sub_assign()`.

**Recommendation** Consider performing the "trimming" step of removing trailing zero coefficients from polynomials in all cases after arithmetic operations. Additionally, consider writing unit tests to catch such potential edge cases.

**Retest Results** Pull Request 112 introduced a function named `truncate_leading_zeros()` which removes the leading zero coefficients of a polynomial. This function is now called prior to returning the result of the arithmetic operations `add()`, `add_assign()`, `sub()`, and `sub_assign()`. As such, this finding has been marked as "Fixed".

| | |
|---|---|
| **Finding** | **Batch Proof Verification Bypass** |
| **Risk** | **High** Impact: High, Exploitability: Medium |
| **Identifier** | NCC-E001741-016 |
| **Status** | Fixed |
| **Category** | Cryptography |
| **Component** | ginger-lib |
| **Location** | `proof-systems/src/darlin/proof_aggregator.rs` |
| **Impact** | A maliciously crafted set of proofs or tampered verification keys may pass the batch (and aggregated) verification procedure. This might allow attackers to tamper with proofs without legitimate users noticing, potentially impacting the trust in the zero-knowledge proof system. |

**Description**

The function `batch_verify_proofs()` in `proof-systems/src/darlin/proof_aggregator.rs` performs batch verification of Proof Carrying Data (PCD) structures consisting of either `FinalDarlin` or `SimpleMarlin` PCDs. To this end, it performs the succinct verification of the PCDs using the verification keys and get their accumulators as a result of a call to `get_accumulators()`, as can be seen in the code excerpt below.

Subsequently, the `batch_verify_proofs()` function checks whether the returned accumulator `accs_g1` (respectively `accs_g2` further below) is empty, in which case it sets the return value `result_g1`(respectively `result_g2`) to `true`.

```
pub fn batch_verify_proofs<G1, G2, D: Digest, R: RngCore>(
    pcds:                   &[GeneralPCD<G1, G2, D>],
    vks:                    &[MarlinVerifierKey<G1::ScalarField,
     ➜ InnerProductArgPC<G1, D>>],
    g1_vk:                  &DLogVerifierKey<G1>,
    g2_vk:                  &DLogVerifierKey<G2>,
    rng:                    &mut R
) -> Result<bool, Option<usize>>
    // ...

    // Do the succinct verification of the PCDs and get their accumulators
    let (accs_g1, accs_g2) = get_accumulators::<G1, G2, D>(pcds, vks, g1_vk,
     ➜ g2_vk)
        .map_err(|e| {
            end_timer!(verification_time);
            e
        })?;

    // Verify accumulators (hard part)
    let result_g1 = if accs_g1.is_empty() {
        true
    } else {
        DLogItemAccumulator::<G1, D>::check_items::<R>(
            g1_vk, &accs_g1, rng
        ).map_err(|_| {
            end_timer!(verification_time);
            None
        })?
    };
```

```
    let result_g2 = if accs_g2.is_empty() {
        true
    } else {

        // ...

    Ok(result_g1 && result_g2)
}
```

The combination of this default "success" return value, together with the vector operations performed in the `get_accumulators()` function, may allow attackers to bypass verification, thereby forging batch or aggregated proofs.

More specifically, the `get_accumulators()` function iterates over its `pcds` and `vks` arguments and performs computations on their respective elements by calling the `zip()` iterator, highlighted in the code excerpt below.

```
pub(crate) fn get_accumulators<G1, G2, D: Digest>(
    pcds:      &[GeneralPCD<G1, G2, D>],
    vks:       &[MarlinVerifierKey<G1::ScalarField, InnerProductArgPC<G1, D>>],
    g1_ck:     &DLogCommitterKey<G1>,
    g2_ck:     &DLogCommitterKey<G2>,
) -> Result<(Vec<DLogItem<G1>>, Vec<DLogItem<G2>>), Option<usize>>
    // ...

    let accs = pcds
        .into_par_iter()
        .zip(vks)
        .enumerate()
        .map(|(i, (pcd, vk))|
            {
                // ...
                pcd.succinct_verify(&vk).map_err(|_| Some(i))
            }
        ).collect::<Result<Vec<_>, _>>().map_err(|e| {
            end_timer!(accumulators_time);
            e
        })?;

    let accs_g1 = accs.iter().flat_map(|acc| acc.0.clone()).collect::<Vec<_>>();
    let accs_g2 = accs.into_iter().flat_map(|acc| acc.1).collect::<Vec<_>>();

    end_timer!(accumulators_time);

    Ok((accs_g1, accs_g2))
}
```

As such, the iteration over the `pcds` and `vks` vectors will stop as soon as one of these vectors is exhausted. Since neither the `get_accumulators()` function, nor the calling `batch_verify_proofs()` function performs any consistency check on the respective lengths of these arrays, a few cases may result in unexpected behavior or potential forgeries.

1. Submitting an empty verification key array (`vks`) to the `batch_verify_proofs()` function successfully returns, regardless of the content of the other parameters, such as the `pcds` array.
2. Submitting an empty proof carrying data array (`pcds`) to the `batch_verify_proofs()` function successfully returns, regardless of the content of the other parameters, such as

the `vks` array.

3. Submitting arrays of different lengths for `pcds` and `vks` to the `batch_verify_proofs()` function returns successfully, provided that the `pcds` and `vks` elements are correct up to the size of the smallest of the two arrays. This might allow an attacker to forge proofs by arbitrarily inflating a valid `pcds` array with invalid proofs.

Note that these comments also apply (to some extent) to the function `verify_aggregated_proofs()`, which also calls the function `get_accumulators()` under the hood, and has a similar default "success" return value.

**Recommendation** Perform strict input validation of all parameters supplied to the functions, in particular when said functions may handle maliciously crafted input. Ensure the lengths of the different vectors are consistent with each other and non-zero.

Additionally, consider revisiting the default assignment of successful return values in the `batch_verify_proofs()` and `verify_aggregated_proofs()` functions.

**Retest Results** Pull Request 112 introduced a validation step in the function `get_accumulators()`, whereby the respective lengths of `pcds` and `vks` are checked to be equal and non-zero, as follows:

```
    if pcds.len() == 0 || vks.len() == 0 || pcds.len() != vks.len() {
        return Err(None);
    }
```

This prevents the verification bypass described above. As such, this finding has been marked as "Fixed".

**Finding** **Incorrect Random Polynomial Generation**

**Risk** **High** Impact: High, Exploitability: Medium

**Identifier** NCC-E001741-017

**Status** Fixed

**Category** Cryptography

**Component** marlin

**Location**
- `marlin/src/ahp/prover.rs`
- `ginger-lib/proof-systems/src/darlin/data_structures.rs`
- `ginger-lib/proof-systems/src/darlin/pcd/mod.rs`

**Impact** The generation of masking polynomials with inadequate random coefficients may invalidate the security proofs and breach the zero-knowledge property.

**Description** As part of the proving procedure performed by Marlin, the function `prover_first_round()` in `marlin/src/ahp/prover.rs` has the ability to mask the polynomials `w_poly`, `z_a_poly` and `z_b_poly` by sampling a random "mask" polynomial in order to achieve zero-knowledge.

Specifically, this function generates a random polynomial from a vector of random elements, conditional on the value of the "zero-knowledge" (`zk`) flag. The first of the three instances is shown in the excerpt below.

```
    // Degree of w_poly before dividing by v_X equals max(|H| - 1 , (zk_bound -
    ➜ 1) + |H|) = (zk_bound - 1) + |H|
    let w_poly = {
        let w = EvaluationsOnDomain::from_vec_and_domain(w_poly_evals,
        ➜ domain_h.clone())
            .interpolate();
        if zk {
            &w + &(&Polynomial::from_coefficients_slice(&vec![F::rand(rng);
            ➜ zk_bound] ) * &v_H)
        } else {
            w
        }
    };
```

However, instead of sampling a random vector of `zk_bound` elements, this construction effectively samples a single random element and duplicates it `zk_bound` times. The same operation is also performed for the polynomials `z_a_poly` and `z_b_poly`.

As a result, the masking polynomials are not random and their efficacy in providing zero-knowledge might be diminished.

On a related note, similar operations are also performed in some `ginger-lib` test code. For example, the generation of random $x_i$s in `ginger-lib/proof-systems/src/darlin/data_structures.rs` also generates a single random element and repeats it `log_key_len_g1` times instead of generating that many random numbers.

```
let random_xi_s_g1 = SuccinctCheckPolynomial::<G1::ScalarField>(vec![u128::rand(
➜ rng).into(); log_key_len_g1 as usize] );
```

Similarly, in `ginger-lib/proof-systems/src/darlin/pcd/mod.rs`, the `simple_marlin.`

`usr_ins` and `final_darlin.usr_ins` vectors will be composed of the same random `G1` element `ins_len` times.

```
    match self {
        Self::SimpleMarlin(simple_marlin) => {
            // No sys ins (for now) for SimpleMarlin, so modify the usr_ins inste
            ➜ ad
            let ins_len = simple_marlin.usr_ins.len();
            simple_marlin.usr_ins = vec![G1::ScalarField::rand(rng); ins_len];
        },
        Self::FinalDarlin(final_darlin) => {
            let ins_len = final_darlin.usr_ins.len();
            final_darlin.usr_ins = vec![G1::ScalarField::rand(rng); ins_len];
        }
    }
```

**Recommendation** Update the random vector generation procedures to produce vectors of distinct random elements, for example by using the `map()` and `collect()` operators on a range, akin to the construction provided below as example.

```
let random_vector: Vec<F> = (0..zk_bound).map(|_| F::rand(rng)).collect();
```

**Retest Results** Pull Request 112 for `ginger-lib` and Pull Request 19 for `marlin` introduced changes to the different random polynomial generations, following the recommended approach. For example in `ginger-lib/proof-systems/src/darlin/data_structures.rs`, the generation of the `random_xi_s_g1` variable is performed as follows:

```
    let random_xi_s_g1 = SuccinctCheckPolynomial::<G1::ScalarField>(
        (0..log_key_len_g1 as usize).map(|_| u128::rand(rng).into()).collect()
    );
```

As such, this finding has been marked as "Fixed".

**Finding** Missing Length Check in Canonical Deserialization

**Risk** **Medium** Impact: Medium, Exploitability: Medium

**Identifier** NCC-E001741-001

**Status** False Positive

**Category** Data Validation

**Component** ginger-lib

**Location** `algebra/src/fields/macros.rs`

**Impact** Different serialized field elements may be deserialized to the same value, resulting in potentially adverse and unexpected consequences, including breach of consensus.

**Description** The function `deserialize_with_flags()` (and the related `deserialize()`) in `macros.rs` deserializes bytes provided as an argument via an implementation of a Rust `Read` trait, as can be seen in the code excerpt below. To read a field element, the `deserialize_with_flags()` function calculates the number of bytes required to represent a field element, and subsequently populates an output buffer by reading exactly that many bytes, highlighted in the code below.

```
impl<P: $params> CanonicalDeserializeWithFlags for $field<P> {
    fn deserialize_with_flags<R: Read, F: Flags>(
        mut reader: R,
    ) -> Result<(Self, F), SerializationError> {
        // All reasonable `Flags` should be less than 8 bits in size
        // (256 values are enough for anyone!)
        if F::BIT_SIZE > 8 {
            return Err(SerializationError::NotEnoughSpace);
        }
        // Calculate the number of bytes required to represent a field element
        // serialized with `flags`. If `F::BIT_SIZE < 8`,
        // this is at most `$byte_size + 1`
        let output_byte_size = buffer_byte_size(P::MODULUS_BITS as usize + F::
        ➜ BIT_SIZE);

        let mut masked_bytes = [0; $byte_size + 1];
        reader.read_exact(&mut masked_bytes[..output_byte_size])?;

        let flags = F::from_u8_remove_flags(&mut masked_bytes[output_byte_size -
        ➜ 1])
            .ok_or(SerializationError::UnexpectedFlags)?;

        Ok((Self::read(&masked_bytes[..])?, flags))
    }
}

impl<P: $params> CanonicalDeserialize for $field<P> {
    fn deserialize<R: Read>(reader: R) -> Result<Self, SerializationError> {
        Self::deserialize_with_flags::<R, EmptyFlags>(reader).map(|(r, _)| r)
    }
}
```

However, due to the nature of the Rust `Read` trait used, this function allows two different inputs to be deserialized to the same field element. Indeed, the function never checks that the totality of the input has been consumed, and byte arrays larger than the expected size are handled without raising any concerns. For example, appending an arbitrary number of bytes after a correctly serialized element produces an equivalent field element upon deserialization. This behavior may lead to unexpected consequences.

**Recommendation** Consider updating the `deserialize_with_flags()` function to return an error if there are more bytes to be read after an element and its flags have been deserialized.

**Retest Results** The client response provided below discusses how the proposed remediation to this finding should be implemented at the application level, since the ability to deserialize data streams of lengths different than that of a field element is leveraged within `ginger-lib`. As such, this finding has been marked as "False Positive".

**Client Response** The customer provided the following response:

> "It's true indeed that, if we want to deserialize a single field element and pass an arbitrary length array, only the first n bytes will be taken into account and the deserialization will be successful; for the same reason, it is also true that we can pass two arbitrary arrays of arbitrary length and, as long as their first n bytes are the same, they will both deserialize successfully to the same field element. However, the proposed solution is not exploitable, as the Field element deserialization function is called by the deserialization function of more complex structs that have many field elements inside. Let's consider the case of a GroupAffine struct (elliptic curve point in affine coordinates): they have two field elements corresponding to the x and y coordinates, and these field elements should be deserialized using the same Read object. While deserializing the x coordinate we cannot enforce that the Read object length is exactly field_element_bytes as this is not true and it's not supposed to be true, since the Read object is used to read an elliptic curve point made out of 2 field elements. The proposed solution should be implemented at application level (mc-cryptolib and sc-cryptolib), where we know the concrete types and their size, and we can check for the overall Read object size.
>
> **In mc-cryptolib:** The Rust-C++ FFI is such that we pass to Rust, pointers to data and how many bytes the data are made up of. Of course, the caller can pass a pointer to arbitrary data of arbitrary length, but this can't be really checked Rust-size: we can only check that the declared data length is equal to the expected one (for fixed size types).
>
> **zend_oo** controls however that the data size is the expected one.
>
> **In sc-cryptolib:** JNI classes always checks that byte buffer sizes are equal to the expected size of the element to be deserialized before deserializing, so it should be fine."

| | |
|---|---|
| **Finding** | **No Domain Separation in Merkle Tree Implementation** |
| **Risk** | **Medium** Impact: High, Exploitability: Medium |
| **Identifier** | NCC-E001741-010 |
| **Status** | Risk Accepted |
| **Category** | Cryptography |
| **Component** | ginger-lib |
| **Location** | `primitives/src/merkle_tree` |
| **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 current Merkle tree implementation in `ginger-lib` does not intrinsically differentiate between *internal* nodes and *leaf* nodes when hashing them. A well-known property of Merkle trees which do not differentiate between internal and leaf nodes is that they lack *second-preimage resistance*: given a root $R$ and tree $T$, it is possible to compute a tree $T'$ that also produces $R$.[1]

A trivial demonstration of this weakness is showcased 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.

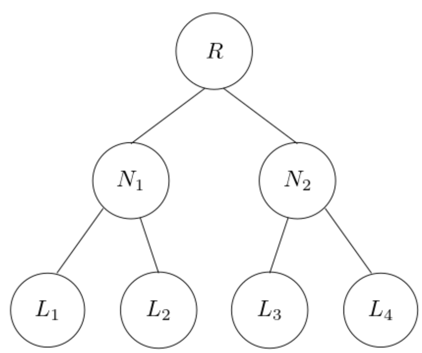


Figure 1: Merkle Tree with four leaves

It is easy to see 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.

[1] https://flawed.net.nz/2018/02/21/attacking-merkle-trees-with-a-second-preimage-attack/

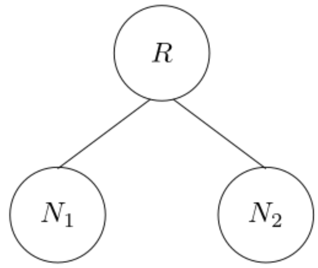


Figure 2: Merkle Tree with two leaves, resulting in the same root

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 Lerner[2] explores this attack in detail: the cost is that of brute-forcing a relatively large search space (between 69 and 73 bits).

In the current Merkle tree implementations, hashing leaves and internal nodes are not being differentiated. For example, consider the functions `hash_inner_node()` and `hash_leaf()` provided below for reference, which both end up with a call to `H::evaluate(parameters, &buffer[..])`, regardless of whether the buffer contains a leaf or two nodes.

```
pub(crate) fn hash_inner_node<H: FixedLengthCRH>(
    parameters: &H::Parameters,
    left: &H::Output,
    right: &H::Output,
    buffer: &mut [u8],
) -> Result<H::Output, Error> {
    use std::io::Cursor;
    let mut writer = Cursor::new(buffer);
    // Construct left input.
    left.write(&mut writer)?;

    // Construct right input.
    right.write(&mut writer)?;

    let buffer = writer.into_inner();
    H::evaluate(parameters, &buffer[..])
}

/// Returns the hash of a leaf.
pub(crate) fn hash_leaf<H: FixedLengthCRH, L: ToBytes>(
    parameters: &H::Parameters,
    leaf: &L,
    buffer: &mut [u8],
) -> Result<H::Output, Error> {
    use std::io::Cursor;
    let mut writer = Cursor::new(buffer);
    leaf.write(&mut writer)?;
```

[2] https://bitslog.com/2018/06/09/leaf-node-weakness-in-bitcoin-merkle-tree-design/

```
    let buffer = writer.into_inner();
    H::evaluate(parameters, &buffer[..])
}
```

Note that this issue might be partially mitigated in this code base by the fact that trees have a fixed height, and that some leaf and node values have fixed (and possibly different) lengths.

**Recommendation** Consider adding a different domain separator in the hash function call when hashing leaves and internal nodes, in order to prevent the kind of second-preimage attack described above.

**Retest Results** With Pull Request 112, the following disclaimer was added to the different Merkle tree implementations (namely, `merkle_tree/field_based_mht/naive/mod.rs`, `merkle_tree/field_based_mht/optimized/mod.rs` and `merkle_tree/mod.rs`) to warn library users about missing domain separation:

```
/// 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.
```

Additionally, an issue on the `ginger-lib` GitHub repository (see Issue 110) was created to track and eventually fix the lack of domain separation. This seems to indicate that the finding will be fixed eventually. As a result, this finding was marked as “Risk Accepted”.

| | |
|---|---|
| **Finding** | **Merkle Leaf Nodes Not Zeroed on Reset** |
| Risk | **Medium** Impact: High, Exploitability: Low |
| Identifier | NCC-E001741-015 |
| Status | Fixed |
| Category | Other |
| Component | ginger-lib |
| Location | `primitives/src/merkle_tree/field_based_mht/optimized/mod.rs` |
| Impact | Incorrect values may be computed for root nodes, subtree nodes, and tree paths. Computed values may not be reproducible between users or between consecutive program executions. |

**Description**

The trait `FieldBasedMerkleTree` specifies a number of methods, including `reset()`. This method is intended to "reset the internal state of the tree, bringing it back to the initial one." The implementation of this method for `FieldBasedOptimizedMHT` is as follows:

```
fn reset(&mut self) -> &mut Self {
    for i in 0..self.new_elem_pos.len() {
        self.new_elem_pos[i] = self.initial_pos[i];
        self.processed_pos[i] = self.initial_pos[i];
    }
    self.finalized = false;

    self
}
```

In this excerpt's main loop, `reset()` resets the indices at which new leaves should be inserted to the tree, effectively ensuring that new leaf values will overwrite old ones; however, it does not perform any leaf zeroing. Under certain conditions, this opens up the possibility for pre-reset leaf values to sneak into post-reset tree evaluations.

If the post-reset tree is not fully saturated, some pre-reset values will be retained in memory, albeit at locations "ahead" of the current insertion index. This becomes problematic when the tree is finalized through `finalize()` or `finalize_in_place()`, as both of these methods begin by moving the leaf insertion index `new_elem_pos[0]` past the end of the leaf buffer. Once this happens, pre-reset leaves are indistinguishable from post-reset leaves, and both will be included in the forthcoming evaluation of the tree.

This will produce incorrect results for nodes at all levels, up to and including the root, as well as for paths through these nodes. This may also have the potential to leak information about the tree's prior contents.

**Recommendation**

Set all leaf nodes to `<T::Data as Field>::zero()` on reset.

**Retest Results**

Pull Request 112 introduced a "zeroization" step in the function `reset()`, whereby every node is overwritten with the zero element, as follows:

```
    // Reset all nodes values
    self.array_nodes.iter_mut().for_each(|leaf| *leaf = <T::Data as
    ➜ Field>::zero());
```

This addresses the issue described above. As such, this finding has been marked as "Fixed".

| | |
|---|---|
| **Finding** | **Incorrect Hiding Bound in Labeled Polynomial Commitment** |
| **Risk** | **Medium** Impact: Medium, Exploitability: Medium |
| **Identifier** | NCC-E001741-022 |
| **Status** | Fixed |
| **Category** | Data Exposure |
| **Component** | poly-commit |
| **Location** | `src/ipa_pc/mod.rs` |
| **Impact** | An incorrect hiding bound in the labeled polynomial to which a commitment is created may result in incorrect computation results, information leakage and loss of zero-knowledge. |

**Description**

In the function `batch_open_individual_opening_challenges()`, the call to the function `commit()` to generate a polynomial commitment fails to specify a hiding bound for the `LabeledPolynomial` to which it commits.

Specifically, the function specifies `None` for the hiding bound regardless of whether or not the `has_hiding` variable was set, as can be seen in the code excerpt below.

```
let (h_commitments, h_randomnesses) = Self::commit(
            &ck,
            vec![&LabeledPolynomial::new(format!("h_poly"), h_polynomial.clone(),
            ➜ None, None)],
            if has_hiding {
                if rng.is_none() {
                    Err(Error::Other("Rng not set".to_owned()))?
                }
                Some(rng.as_mut().unwrap())
            } else {
                None
            }
        )?;
```

In comparison, the creation of such a polynomial is performed correctly a few lines below (starting on line 1393), as can be seen in the following code excerpt.

```
        let labeled_batch_polynomial = LabeledPolynomial::new(
            format!("LC"),
            lc_polynomial,
            None,
            if has_hiding { Some(1) } else { None }
        );
```

**Recommendation**

Specify the correct value for the `hiding_bound` parameter. For example, by creating the labeled polynomial using the following approach.

```
    vec![&LabeledPolynomial::new(format!("h_poly"), h_polynomial.clone(), None,
    ➜  if hiding_bound { Some(1) } else { None })],
```

**Retest Results**

In a recent commit (Fix missing hiding bound on h_poly commitment), the solution recommended above was implemented. As such, this finding has been marked as “Fixed”.

| | |
|---|---|
| **Finding** | **Secure Rust Best Practices Not Always Followed** |
| **Risk** | **Low** Impact: Medium, Exploitability: Low |
| **Identifier** | NCC-E001741-002 |
| **Status** | Partially Fixed |
| **Category** | Other |
| **Component** | ginger-lib |
| **Location** | Systemic |
| **Impact** | Good programming practices ensure that bugs and vulnerabilities are less likely to be introduced in the code base and easier to identify when they occur, and also help code maintainability. For example, exceptional conditions which cause an unhandled panic may present a denial of service vector. |

**Description**

While overall good programming practices were observed throughout the different code bases, the NCC Group team observed a few instances of less-than-ideal Rust programming practices, mostly around error handling.

The Rust programming language provides specific constructions representing return values, in the form of the `Option` and the `Result` enums. These values provide the ability to both represent a successful result and the possibility of an empty return value (or an error, respectively). In order to access the underlying result, the function `unwrap()` may be used. This function returns the result of the function, but panics if there was an error. Its use can be justified in some cases, but blindly using the `unwrap()` function as a shortcut way to obtain the result can lead to issues up the calling stack, and obscure the underlying problems. The NCC Group team noted that the use of `unwrap()` was widespread throughout the code base.

Another example of a typical pattern that may lead to denial of service conditions is the usage of the `assert!()` macro, which also results in panics if not fulfilled.

Generally speaking, explicit error handling should be preferred instead of calling functions that might result in panics, such as `unwrap()` or `expect()`. The Secure Rust Guidelines provide some helpful pointers to that effect.

Finally, another helpful tool to assess the adherence of a code base to best practices is the `cargo clippy` utility. Running that tool on the various code repositories showed a number of constructions that could be improved upon. As an example, running `cargo clippy` on the current `ginger-lib` repository results in more than 800 emitted warnings.

**Recommendation**

Consider performing a pass throughout all code bases and converting `unwrap()` and `assert!()` calls to more explicit error handling.

Add a gating milestone to the development process that involves running `cargo clippy` and fixing the emitted warnings.

**Retest Results**

In a series of five Pull Requests (one for each library in scope, see below) the Horizen Labs team made a concentrated effort at better following Rust secure programming practices.

- `ginger-lib`: PR 118
- `marlin`: PR 24
- `poly-commit`: PR 26
- `zendoo-cctp-lib`: PR 23

- `zendoo-mc-cryptolib`: PR 48

More specifically, a large number of `unwrap()` calls were removed and panic-inducing construction were converted to safer code by making use of the Rust `Result` and `Option` types. A number of `assert()` calls were also removed and comments were added whenever the use of `unwrap()` was safe.

The team also indicated that the introduction of the tools `cargo clippy` and `cargo fmt` was going to be introduced to the development process at a later stage. Given the fact that following best programming practices is an ongoing effort, this finding was marked as "Partially Fixed".

| | |
|---|---|
| **Finding** | **Misleading Modular Reduction Function** |
| **Risk** | **Low** Impact: Medium, Exploitability: Low |
| **Identifier** | NCC-E001741-004 |
| **Status** | Fixed |
| **Category** | Cryptography |
| **Component** | ginger-lib |
| **Location** | `algebra/src/fields/macros.rs` |
| **Impact** | API misuse due to misleading function naming may result in incorrect and unexpected results. |

**Description**

The file `algebra/src/fields/macros.rs` implements a number of operations on arbitrary finite field elements. Among these operations, the `reduce()` function is used to compute $z$ mod $n$, namely to reduce an element modulo the field order $n$ (`P::MODULUS` in the code excerpt below).

```
fn reduce(&mut self) {
    if !self.is_valid() {
        self.0.sub_noborrow(&P::MODULUS);
    }
}
```

The `reduce()` function is not complete when reducing elements. Namely, if the element is invalid (i.e., larger that the modulus) it only subtracts the field modulus from the element once, and assumes the element to be reduced after that. As such, values larger than (or equal to) $2n$ will not be correctly reduced.

Luckily, this seems inconsequential since `reduce()` currently appears to be called to reduce elements that cannot be larger that twice the modulus, by design. For example, the functions `double_in_place()` and `add_assign()` in `algebra/src/fields/macros.rs` both call `reduce()` with valid values.

Nevertheless, this may pose a risk to developers and future unsuspecting users of this library.

**Recommendation**

Consider implementing a more complete modular reduction routine, such as Barrett[3] or Montgomery[4] reduction.

At the very least, consider adding code comments clearly outlining the limitations of this function, in order to prevent developers from calling it when expecting a full modular reduction.

**Retest Results**

With Pull Request 112, a comment was added to the `reduce()` function indicating that its behavior is correct if and only if the value to reduce is smaller or equal than twice the field modulus. This is in line with the second recommendation provided above. As such, this finding has been marked as "Fixed".

[3] https://en.wikipedia.org/wiki/Barrett_reduction
[4] https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

| | |
|---|---|
| **Finding** | **Potential Panic with Zero-Division** |
| **Risk** | **Low** Impact: Medium, Exploitability: Low |
| **Identifier** | NCC-E001741-005 |
| **Status** | Fixed |
| **Category** | Data Validation |
| **Component** | ginger-lib |
| **Location** | `algebra/src/fields/macros.rs` |
| **Impact** | Missing validation checks that result in a panic may present a denial of service attack vector. |

**Description**

The `div()` function located in the file `macros.rs` implements the field division operation. To do so, it computes the inverse of the divisor passed in as argument (the `other` variable in the code excerpt below) and multiplies the dividend by the result of this field inversion.

```
#[inline]
fn div(self, other: &Self) -> Self {
    let mut result = self.clone();
    result.mul_assign(&other.inverse().unwrap());
    result
}
```

However, the `div()` function does not check that the `other` parameter is non-zero. As such, trying to divide by zero will result in a panic. Indeed, the function `inverse()` returns a Rust `Option` type. When trying to compute the inverse of the zero-element, `inverse()` returns `None`, which will panic when `unwrap`-ed.

**Recommendation**

Gracefully handle division by zero so as not to panic upon unexpected inputs. Consider updating the `div()` function to return a Rust `Result` or `Option` type, similar to the `inverse()` function.

**Retest Results**

In the same series of Pull Requests (one for each library in scope, see below) addressing finding NCC-E001741-002 on page 20, the Horizen Labs team made a concentrated effort to prevent potential instances of divisions by zero earlier in the call hierarchy, in the different callers of the `div()` and `div_assign()` functions.

- `ginger-lib`: PR 118
- `marlin`: PR 24
- `poly-commit`: PR 26
- `zendoo-cctp-lib`: PR 23
- `zendoo-mc-cryptolib`: PR 48

As a result, this finding was marked as "Fixed".

| | |
|---|---|
| **Finding** | **Outdated and Vulnerable Rust Dependencies** |
| **Risk** | **Low** Impact: Medium, Exploitability: Low |
| **Identifier** | NCC-E001741-006 |
| **Status** | Fixed |
| **Category** | Patching |
| **Component** | ginger-lib |
| **Location** | `ginger-lib` |
| **Impact** | An attacker may attempt to identify and utilize vulnerabilities in outdated dependencies to exploit the application. |

**Description**

Using outdated dependencies with discovered vulnerabilities is one of the most common and serious routes of application exploitation. Many of the most severe breaches have relied upon exploiting known vulnerabilities in dependencies.[5]

Some convenient tools exist to assess the health of the dependencies of Rust code bases. The utility `cargo-audit`[6] audits `Cargo.lock` *files for crates with security vulnerabilities reported to the RustSec Advisory Database*.

Running the tool `cargo audit` on the `ginger-lib` directory shows that one vulnerability exists in the `blake2` crate, and also highlights some warnings related to unmaintained and potentially vulnerable dependencies. An excerpt of its output is provided below for reference.

```
$ cargo audit

error: Vulnerable crates found!

ID:       RUSTSEC-2019-0019
Crate:    blake2
Version:  0.7.1
Date:     2019-08-25
URL:      https://rustsec.org/advisories/RUSTSEC-2019-0019
Title:    HMAC-BLAKE2 algorithms compute incorrect results
Solution:  upgrade to >= 0.8.1
Dependency tree:
blake2 0.7.1

warning: 2 warnings found

Crate:  dirs
Title:  dirs is unmaintained, use dirs-next instead
Date:   2020-10-16
URL:    https://rustsec.org/advisories/RUSTSEC-2020-0053
Dependency tree:
dirs 1.0.5
└── term 0.5.2
    └── clippy 0.0.302
        └── algebra 0.1.0
```

[5] https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/
[6] https://github.com/RustSec/cargo-audit

```
Crate:  term
Title:  term is looking for a new maintainer
Date:   2018-11-19
URL:    https://rustsec.org/advisories/RUSTSEC-2018-0015
Dependency tree:
term 0.5.2
└── clippy 0.0.302
    └── algebra 0.1.0

Crate:   marlin
Version:  0.1.0
Warning:  package has been yanked!
Dependency tree:
marlin 0.1.0
└── proof-systems 0.1.0
    └── r1cs-crypto 0.1.0
        ├── r1cs-crypto 0.1.0
        └── proof-systems 0.1.0

error: 1 vulnerability found!
warning: 2 warnings found!
```

Another interesting tool is the `cargo-outdated`[7] utility, which is a *cargo subcommand for displaying when Rust dependencies are out of date.* An excerpt of the output of running the `cargo outdated` subcommand on the `ginger-lib` repository is provided below. A number of dependencies are outdated.

```
algebra
================
Name              Project  Compat  Latest   Kind         Platform
----              -------  ------  ------   ----         --------
blake2            0.7.1    ---     0.9.1    Development  ---
byte-tools        0.2.0    ---     Removed  Normal       ---
cfg-if            1.0.0    ---     Removed  Normal       ---
colored           1.9.3    ---     2.0.0    Normal       ---
constant_time_eq  0.1.5    ---     Removed  Normal       ---
crypto-mac        0.5.2    ---     0.8.0    Normal       ---
digest            0.7.6    ---     0.9.0    Normal       ---
generic-array     0.9.1    ---     0.14.4   Normal       ---
getrandom         0.1.16   ---     0.2.3    Normal       ---
getrandom         0.1.16   ---     Removed  Normal       ---
itertools         0.10.0   0.10.1  0.10.1   Normal       ---
libc              0.2.95   0.2.97  0.2.97   Normal       ---
libc              0.2.95   0.2.97  0.2.97   Normal       cfg(unix)
libc              0.2.95   0.2.97  Removed  Normal       cfg(unix)
rand              0.7.3    ---     0.8.4    Normal       ---

// ...
```

**Recommendation** Update all dependencies and tools to the latest versions recommended for production deployment. Add a gating milestone to the development process that involves reviewing all dependencies for outdated or vulnerable versions.

**Retest Results** In a series of five Pull Requests (one for each library in scope, see below) the Horizen Labs team updated all outdated dependencies to their latest compatible versions.

[7] https://github.com/kbknapp/cargo-outdated

- `ginger-lib`: PR 112
- `marlin`: PR 21
- `poly-commit`: PR 20
- `zendoo-cctp-lib`: PR 21
- `zendoo-mc-cryptolib`: PR 45

Additionally, cargo-audit was added to the continuous integration development process. As a result, this finding was marked as "Fixed".

**Finding** **Insufficient Parameter Checks in Multi-Scalar Multiplication**

**Risk** **Low** Impact: Medium, Exploitability: Low

**Identifier** NCC-E001741-008

**Status** Fixed

**Category** Data Validation

**Component** ginger-lib

**Location** `algebra/src/msm/variable_base.rs`

**Impact** Insufficient parameter validation is one of the most common cause of software vulnerabilities, which can lead to undesired and unexpected behavior, or system crashes in some cases.

**Description** Given the group elements $G_1, \ldots, G_n$ of a cyclic group and the integers $a_1, \ldots, a_n$ between 0 and the group order, multi-scalar multiplication is known as the problem of computing the group element $a_1G_1 + \ldots + a_nG_n$.

The `ginger-lib` repository provides different naive and efficient implementations of multi-scalar multiplication in its `algebra/src/msm` subdirectory. The NCC Group team observed a few instances within this directory where missing parameter validation could lead to undesired and unexpected behavior.

Specifically, the functions `multi_scalar_mul_affine_c()` and `msm_inner_c()` (see the signature of the former below and note that the latter has the same signature) do not perform any validation of their parameters.

```
8  pub fn multi_scalar_mul_affine_c<G: AffineCurve>(
9          bases: &[G],
10         scalars: &[<G::ScalarField as PrimeField>::BigInt],
11         c: usize
12     ) -> G::Projective
13
14         let cc = 1 << c;
15
16         let num_bits =
17             <G::ScalarField as PrimeField>::Params::MODULUS_BITS as usize;
18         let fr_one = G::ScalarField::one().into_repr();
19
20         let zero = G::zero().into_projective();
21         let window_starts: Vec<_> = (0..num_bits).step_by(c).collect();
```

This may result in several unexpected issues.

First, passing a value of 0 for the parameter `c` (the window size) to either of these functions will result in a Rust panic. This is due to the call to `step_by()` on line 21 highlighted above, which panics on a failed assertion that the step is non-zero (`panicked at 'assertion failed: step != 0'`).

Second, there is no upper-bound check on this variable `c`. This may lead to large amounts of memory being allocated, since both functions declare a `buckets` vector of size approximately $2^c$, as can be seen on line 107 of the `msm_inner_c()` function:

```
    let mut buckets = vec![zero; (1 << c) - 1];
```

An attacker may be able to impede the normal behavior of processes if they were able to influence the value of that variable.

Third, the functions do not perform any validation on the respective lengths of the `bases` and `scalars` parameters. More specifically, the computation still succeeds with different lengths for the scalars and base points vectors, simply discarding the superfluous elements. This may lead to potential message malleability issues. For example, consider a legitimate multi-scalar multiplication with scalars $a_1, a_2$ and group elements $G_1, G_2$ resulting in the group element $G$. An attacker submitting an inflated scalar list consisting of $a_1, a_2, a_3$ (but with the same group elements $G_1, G_2$) will obtain the same final group element $G$ as the original list above.

**Recommendation** Consider performing stricter parameter validation in all multi-scalar multiplication-related functions. The examples listed above should not be considered exhaustive and other instances where insufficient parameter validation leads to errors might exist within the repository.

Additionally, since the function `msm_inner_c()` seems to be exclusively called by its wrapper `msm_inner()` which computes an optimal window size, consider changing its visibility[8] (currently `pub`) so that it cannot be called from outside of the crate.

**Retest Results** Pull Request 112 introduced sanitation checks in the form of assertions on the variable `c` and on the lengths of the scalars and bases arrays, for the two functions `multi_scalar_mul_affine_c()` and `msm_inner_c()`, as follows:

```
// Sanity checks
assert!(c != 0, "Invalid window size value: 0");
assert!(c <= 25, "Invalid window size value: {}. It must be smaller than 25",
 ➜ c);
assert!(
    scalars.len() <= bases.len(),
    "Invalid MSM length. Scalars len: {}, Bases len: {}", scalars.len(),
     ➜ bases.len()
);
```

Since the fact that `scalars.len()` may be shorter than `bases.len()` in some concrete cases, a disclaimer was added, calling out the malleability issue specifically:

```
  /// WARNING: This function allows scalars and bases to have different length
  /// (as long as scalars.len() <= bases.len()): internally, bases are trimmed
  /// to have the same length of the scalars; this may lead to potential messag
   ➜ e
  /// malleability issue: e.g. MSM([s1, s2], [b1, b2]) == MSM([s1, s2], [b1, b2
   ➜ , b3]),
  /// so use this function carefully.
```

This is in line with the recommendations provided above. As such, this finding has been marked as "Fixed".

[8] https://doc.rust-lang.org/reference/visibility-and-privacy.html

| | |
|---|---|
| **Finding** | **Insufficient Parameter Validation in Merkle Tree Implementation** |
| **Risk** | **Low** Impact: Medium, Exploitability: Low |
| **Identifier** | NCC-E001741-011 |
| **Status** | Fixed |
| **Category** | Data Validation |
| **Component** | ginger-lib |
| **Location** | `primitives/src/merkle_tree/mod.rs` |
| **Impact** | Insufficient parameter validation is one of the most common cause of software vulnerabilities, which can lead to undesired and unexpected behavior, or system crashes in some cases. |

**Description** The `primitives/src/merkle_tree` repository implements different variants of Merkle trees, from naive to optimized versions, as well as structures to deal with paths within the trees. The NCC Group team noticed a few instances where insufficient parameter validation could lead to unexpected behavior or Rust panics.

- The creation of Merkle trees with a single leaf element leads to panics in some cases. Specifically, in the generic implementation `MerkleHashTree` in `primitives/src/merkle_tree/mod.rs`, the function `new()` makes use of the `next_power_of_two()`, which returns one when presented with an array of size one. Later in this function, there is an array access at an index larger than the number of elements in the array `tree`, which triggers a panic. Selected portions are highlighted in the code excerpt below.

```
pub fn new<L: ToBytes>(
    parameters: Rc<<P::H as FixedLengthCRH>::Parameters>,
    leaves: &[L],
) -> Result<Self, Error> {
    let new_time = start_timer!(|| "MerkleTree::New");

    let last_level_size = leaves.len().next_power_of_two();
    let tree_size = 2 * last_level_size - 1;
    let tree_height = tree_height(tree_size);
    // ...

    // Compute and store the hash values for each leaf.
    let last_level_index = level_indices.pop().unwrap();
    let mut buffer = vec![0u8; P::H::INPUT_SIZE_BITS/8];
    for (i, leaf) in leaves.iter().enumerate() {
        tree[last_level_index + i] = hash_leaf::<P::H, _>(&parameters, leaf,
        ➜ &mut buffer)?;
    }
```

  Note that similar behavior related to panics at out-of-bound array accesses is also present in the `append()` function in the naive Merkle tree implementation in `primitives/src/merkle_tree/field_based_mht/naive/mod.rs`.

- The function `get_merkle_path()` in `primitives/src/merkle_tree/field_based_mht/optimized/mod.rs` does not check the validity of its `leaf_index` input parameter. Given an out-of-bound value, this will eventually trigger a panic on the array access highlighted in the code excerpt below. This is because the `start_position` variable is initialized with a value equal to `leaf_index` (possibly minus a small integer depending on the arity of the tree).

```
fn get_merkle_path(&self, leaf_index: usize) -> Option<Self::MerklePath> {

// ...

  // We must save the siblings of the actual node
  for i in start_position..end_position {
      if i != node_index {
          siblings.push(self.array_nodes[i])
      }
  }
```

- Currently, no bounds check is performed on the length of the data present in the leaves. The validity of the leaves data seems to rely on the assumption that the underlying instantiations of the macro `array_bytes!` (in `algebra/src/bytes.rs` which implement the trait `ToBytes`) are currently only declared up to 32 bytes. As such, only data up to 32 bytes can be added to a leaf, although instantiations of the `array_bytes!` macro with larger values would increase this upper bound. This implicit bound might lead to issues when relied upon, if the underlying `algebra` repository were to be modified.

**Recommendation** Consider performing stricter parameter validation in all Merkle tree-related functions. The examples listed above should not be considered exhaustive and other instances where insufficient parameter validation leads to errors might exist within the `merkle_tree` repository.

**Retest Results** Pull Request 112, introduced a number of additional validation checks to the different Merkle tree-related functions. These changes now address the first two points discussed in this finding (the third item was not deemed to be a significant issue by the Horizen Labs team). Additionally, extensive unit tests covering edge cases were added as part of the Pull Request. As such, this finding has been marked as “Fixed”.

**Finding** Potential DoS via Memory Exhaustion in Merkle Tree Instantiation

**Risk** Low Impact: Medium, Exploitability: Low

**Identifier** NCC-E001741-012

**Status** Risk Accepted

**Category** Denial of Service

**Component** ginger-lib

**Location** `primitives/src/merkle_tree/mod.rs`

**Impact** An adversary may trigger the allocation of large amounts of memory, eventually impeding the normal behavior of processes.

**Description** The creation of a new Merkle tree may be a potential memory exhaustion vector. Upon creation of a new tree from a list of leaves, the entire tree (including all intermediate nodes) is created and stored in memory, as can be seen in the `new()` function, provided below for reference.

```
111 pub fn new<L: ToBytes>(
112         parameters: Rc<<P::H as FixedLengthCRH>::Parameters>,
113         leaves: &[L],
114     ) -> Result<Self, Error> {
115         let new_time = start_timer!(|| "MerkleTree::New");
116 
117         let last_level_size = leaves.len().next_power_of_two();
118         let tree_size = 2 * last_level_size - 1;
119         let tree_height = tree_height(tree_size);
120         assert!(tree_height as u8 <= Self::HEIGHT);
121 
122         // Initialize the merkle tree.
123         let mut tree = Vec::with_capacity(tree_size);
124         let empty_hash = hash_empty::<P::H>(&parameters)?;
125         for _ in 0..tree_size {
126             tree.push(empty_hash.clone());
127         }
128 
129         // ...
```

As an example, consider the creation of a new tree with a list of $2^n + 1$ leaves. This will result in the creation of a total of $2^{n+2} - 1$ nodes (see lines 117-118 above). For each of these nodes, the default empty hash is copied. Assuming a concrete instantiation of hash functions used with this Merkle tree in which hashes are 64 bytes long, this would result in a total size of $2^{n+8}$, namely a factor of around $2^8$ increase compared to the initial number of leaves.

Depending on the use case and whether this naive tree implementation is publicly exposed, attackers may have the ability to consume large amounts of memory on a target platform.

Note that this applies to some extent to the Optimized Merkle tree variant (`FieldBasedOptimizedMHT`) defined in `primitives/src/merkle_tree/field_based_mht/optimized/mod.rs`, where the `init()` function populates all the nodes with `zero()` values upon creation.

```
    // Initialize to zero all tree nodes
    let mut array_nodes = Vec::with_capacity(tree_size);
```

```
    for _i in 0..tree_size {
        array_nodes.push(<T::Data as Field>::zero());
    }
```

**Recommendation** Consider updating the Merkle tree implementations to use less memory upon creation, for example by removing the copy of the empty hash to all the nodes in the naive case. Alternatively, ensure only small trees may be created by imposing limits on the number of leaves (which directly correlates to the resulting tree size).

**Retest Results** With Pull Request 112, the following disclaimer was added to the different Merkle tree implementations (namely, `merkle_tree/field_based_mht/naive/mod.rs`, `merkle_tree/field_based_mht/optimized/mod.rs` and `merkle_tree/mod.rs`) to warn library users about potentially large memory usage:

```
/// 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.
```

Since a denial of service attack via memory exhaustion cannot be completely ruled out (specifically for other users of the `ginger-lib` library), this finding was marked as "Risk Accepted".

| | |
|---|---|
| **Finding** | **Incoherence in Poseidon Round Number Parameters** |
| **Risk** | **Low** Impact: Low, Exploitability: Low |
| **Identifier** | NCC-E001741-013 |
| **Status** | False Positive |
| **Category** | Cryptography |
| **Component** | ginger-lib |
| **Location** | • `primitives/src/crh/poseidon/parameters/tweedle_dee.rs`<br>• `primitives/src/crh/poseidon/parameters/tweedle_dum.rs` |
| **Impact** | Non-conformance to the cryptographic literature may limit interoperability, or in the worst case, decrease the claimed security guarantee of the primitive. |

**Description**

The Poseidon hash function[9] 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 and last $R_f$ rounds have full S-box layers, while the $R_P$ intermediate rounds only have 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 `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.
```

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 instantiations,[10] 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

[9] https://eprint.iacr.org/2019/458.pdf
[10] https://extgit.iaik.tugraz.at/krypto/hadeshash/-/blob/master/code/calc_round_numbers.py

security of the hash function itself. However, it has the potential to introduce interoperability issues.

**Recommendation** Consider clearly documenting the choice for the variable $R_P$, and where this value originates from, in order to prevent any possible discrepancy between this implementation and concurrent instantiations of Poseidon with the Tweedle curves.

**Retest Results** 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. This is detailed in the *Client Response* below. As such, this finding has been marked as "False Positive"

**Client Response** The customer provided the following response:

> "After having reached out to the Poseidon authors about the inconsistency between the script and the paper, it was clarified that the value of the script is correct."

**Finding** RNG Implementation Non-Compliant with Rust Documentation

**Risk** Low Impact: Low, Exploitability: Low

**Identifier** NCC-E001741-014

**Status** Fixed

**Category** Cryptography

**Component** poly-commit

**Location** `poly-commit/src/rng.rs`

**Impact** Failure to follow requirements imposed by the underlying Rust traits may result in the generation of poor random numbers due to API misuse and potential panics.

**Description** The file `poly-commit/src/rng.rs` provides custom traits and implementations of a pseudo-random number generator, (`FiatShamirRng` and `FiatShamirChaChaRng`, respectively), which are used to derive pseudo-random challenges deterministically for the Darlin proof system.

This implementation fails to satisfy some of the requirements imposed by the underlying Rust RNG traits, such as `SeedableRng`.[11] More specifically, the Rust documentation for the `from_seed()` required method imposes some constraints on the quality of the seed it is instantiated with. Additionally, it also mandates that implementations of this function should never panic. These two points are highlighted in the excerpt of the Rust documentation provided below.

```
    /// Create a new PRNG using the given seed.
    ///
    /// PRNG implementations are allowed to assume that bits in the seed are
    /// well distributed. That means usually that the number of one and zero
    /// bits are roughly equal, and values like 0, 1 and (size - 1) are unlikely.
    /// Note that many non-cryptographic PRNGs will show poor quality output
    /// if this is not adhered to. If you wish to seed from simple numbers, use
    /// `seed_from_u64` instead.
    ///
    /// ...
    ///
    /// PRNG implementations should make sure `from_seed` never panics. In the
    /// case that some special values (like an all zero seed) are not viable
    /// seeds it is preferable to map these to alternative constant value(s),
    /// for example `0xBAD5EEDu32` or `0x0DDB1A5E5BAD5EEDu64` ("odd biases? bad
    /// seed"). This is assuming only a small number of values must be rejected.
    fn from_seed(seed: Self::Seed) -> Self;
```

The custom Fiat-Shamir RNG implementation fails to comply to these two statements. For example, the `new()` function instantiates an RNG using an all-zero seed. Additionally, the `from_seed()` function may panic on malformed input.

```
    fn new() -> Self {
        let seed = [0u8; 32];
        Self::from_seed(&to_bytes![seed].unwrap())
    }

    // ...
```

[11] https://docs.rs/rand/0.6.0/rand/trait.SeedableRng.html

```
/// Create a new `Self` by initializing with a fresh seed.
#[inline]
fn from_seed<'a, T: 'a + ToBytes>(seed: &'a T) -> Self {
    let mut bytes = Vec::new();
    seed.write(&mut bytes).expect("failed to convert to bytes");
    let seed = D::digest(&bytes);
    let r_seed: [u8; 32] =
    ➔ FromBytes::read(seed.as_ref()).expect("failed to get [u32; 8]");
    let r = ChaChaRng::from_seed(r_seed);
    Self {
        r,
        seed,
        digest: PhantomData,
    }
}
```

**Recommendation** The current code base does not seem to misuse the `new()` function and directly use the RNG seeded with all-zero bytes (without reseeding immediately after). However, potential users of this library might fail to follow the same pattern. Hence, assess whether exposing a `new()` function initializing the RNG with an all-zero seed is necessary, and if not, consider removing it.

Additionally, consider updating the `from_seed()` implementation to avoid the potential for panics.

**Retest Results** In Pull Request 112, the `new()` function was removed and the `from_seed()` function was updated such that it cannot panic. This is in line with the recommendations provided above and this finding has been marked as "Fixed" as a result.

**Finding** Ambiguous Fiat-Shamir Oracle Instantiation and Input Serialization

**Risk** Low Impact: Medium, Exploitability: Low

**Identifier** NCC-E001741-018

**Status** Fixed

**Category** Cryptography

**Component** Systemic

**Location**
- ginger-lib/proof-systems/src/darlin/accumulators/dlog.rs
- marlin/src/lib.rs
- poly-commit/src/ipa_pc/mod.rs

**Impact** Oracle input values may be reused with different parameter configurations, leading to the same output and contradicting the random oracle model on which the security proofs are built.

**Description** The *Fiat-Shamir* Random Number Generator (RNG), defined in the file poly-commit/src/rng.rs, is used to derive pseudo-random challenges deterministically, which underly the non-interactive zero-knowledge proof system implemented in Darlin. The general principle of this construction goes as follows: it starts by initializing the RNG from a seed (see the first code excerpt below, from marlin/src/lib.rs), after which it absorbs an arbitrary number of inputs, and finally, a random element is obtained by calling the squeeze_128_bits_challenge() function (see the second code listing below, excerpted from poly-commit/src/ipa_pc/mod.rs).

```
let mut fs_rng = PC::RandomOracle::from_seed(
    &to_bytes![&Self::PROTOCOL_NAME, pc_pk.get_hash(), &index_pk.index_vk,
     ➜ &public_input].unwrap(),
);
```

```
// Absorb evaluations
fs_rng.absorb(&values.iter().flat_map(|val|
 ➜ to_bytes!(val).unwrap()).collect::<Vec<_>>());

// Sample new batching challenge
let random_scalar: G::ScalarField = fs_rng.squeeze_128_bits_challenge();
```

The NCC Group team noted that the length of the different arrays being absorbed are not injected into the Fiat-Shamir RNG (either via the from_seed() or the absorb() function) and there are no extra separators that differentiate the various kinds of elements, which may result in different inputs producing the same hash output.

Conceptually, given the byte array b = [1, 2, 3, 4], the calls absorb(b[0], &b[1..4]), absorb(&b[0..2], &b[2..4]) and absorb(&[1, 2, 3, 4]) are all equivalent. Thus, the overall input to the hash function is ambiguous and different instances of the protocol may use the oracle with the same input string. This implies that the security proofs described in the different reference papers for each protocol may no longer cover the current implementation.

This remark applies both to the instantiation of the oracle using the from_seed() function as well as the absorption using the absorb() function.

Note however that the exploitability of this finding is somewhat mitigated by the fact that the absorb function hashes the concatenation of its old seed with its inputs (i.e., $seed = H(seed||inputs)$) and then returns a new RNG instance from the newly computed seed. Nevertheless, the importance of the Fiat-Shamir construction in the different protocols as well as the ability for attackers to supply inputs to these instances may still result in vulnerabilities.

**Recommendation** Consider prepending the respective lengths of the array inputs to the oracle calls.

**Retest Results** Pull Request 27 introduced changes to the Fiat-Shamir RNG in which the length of the input is prepended to the inputs themselves prior to hashing, which followed the recommended approach. As a result, this finding was marked as "Fixed".

| | |
|---|---|
| **Finding** | **Discrepancy with Reference Paper on Random Challenge Domain** |
| **Risk** | **Low** Impact: Low, Exploitability: Low |
| **Identifier** | NCC-E001741-019 |
| **Status** | False Positive |
| **Category** | Cryptography |
| **Component** | marlin |
| **Location** | `src/ahp/verifier.rs` |
| **Impact** | Implementation discrepancies with the academic references may invalidate the security proofs and breach security guarantees. |

**Description**

This finding describes two distinct discrepancies between the current implementation and the two[12, 13] reference papers regarding the sampling domain of some random challenges.

**1. The Darlin** reference paper imposes specific domain constraints when sampling random elements, such as on page 6, where the random challenge $z$ is sampled from $F \setminus H$:

> $z \xleftarrow{\$} F \setminus H$ (...) (The oracle aborts, if $z \in H$.)

and on page 9, where $\alpha$ is also sampled from the same set;

> $\alpha \leftarrow \$F \setminus H.$

However, in the current implementation, there does not seem to be any domain restriction on the random numbers. Namely, all challenges are obtained from calls to the `squeeze_128_bits_challenge()` function defined in `poly-commit/src/rng.rs`.

**2. The Marlin** reference paper refers to three random elements, $\eta_A, \eta_B, \eta_C$, which are used to bundle three sumcheck into one:

> Next, [**V**erifier] samples random elements $\alpha, \eta_A, \eta_B, \eta_C \in F$ and sends them to [**P**rover]. The element $\alpha$ is used to reduce lincheck problems to sumcheck, while the elements $\eta_A, \eta_B, \eta_C$ are used to bundle the three sumcheck problems into one.

However the current implementation sets the vector $(\eta_A, \eta_B, \eta_C)$ to $(1, \eta, \eta^2)$, as can be seen in the code excerpt below, from `src/ahp/verifier.rs`.

```
let eta: F = fs_rng.squeeze_128_bits_challenge();
let eta_a = F::one();
let eta_b = eta;
let eta_c = eta_b * &eta;
```

The NCC Group team noted that this choice is actually consistent with the Darlin paper, which states that

> *(We notice that using the powers of $\eta$ slightly differs from choosing arbitrary random scalars $\eta_A, \eta_B, \eta_C$ as in [CHM+20], but this does not affect security.)*

However, the claim that security is not affected is not further substantiated in the paper.

[12] *Darlin: Recursive Proofs using Marlin* https://eprint.iacr.org/2021/930
[13] *Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS* https://eprint.iacr.org/2019/1047

| | |
|---|---|
| Recommendation | Consider updating the random challenge generation procedures such that challenges are sampled from the same, restricted set as in the paper. Additionally, consider providing an argument as to why security is not affected by the reduced randomness used in the Marlin reduction. |
| Retest Results | The Horizen Labs team pointed out that the domain constraints were indeed correctly enforced when necessary during the sampling of random elements. Confusion arose due to discrepancies between the reference paper and the implementation in the naming of some variables, see the Client Response field below.<br><br>Additionally, the team indicated that the final version of the reference paper would include a rigorous security analysis regarding replacing multiple random challenges with powers of a single one, which is a well-known technique currently widely used in "second-wave" SNARKs. As a result, this finding was marked as "False Positive". |
| Client Response | The customer provided the following response:<br><br>"In Marlin we already enforce, where necessary, the challenges to be sampled from the correct FFT subdomain. In some cases, namings from the paper are different from the ones inside the code, thus generating such misunderstanding. Regarding replacing multiple random challenges with powers of a single one, we followed a technique also applied in "second wave" SNARKS: for example, Sonic, Halo, or Halo Infinite and the proofs therein. In any case, a rigorous security analysis will be given in the full version of the paper." |

| | |
|---|---|
| **Finding** | **Undefined Behavior in Foreign Function Interface** |
| **Risk** | **Low** Impact: Undetermined, Exploitability: Low |
| **Identifier** | NCC-E001741-021 |
| **Status** | Fixed |
| **Category** | Error Reporting |
| **Component** | zendoo-mc-cryptolib |
| **Location** | Throughout `lib.rs` |
| **Impact** | Undefined behavior may be triggered in foreign code. |

**Description**

The `zendoo-mc-cryptolib` repository exposes a Foreign Function Interface to `ginger-lib` which can be invoked by foreign languages supporting the C ABI. Many of the functions exposed through this interface will `panic!` if they receive unexpected inputs (e.g. null pointers).

Regarding this situation, Chapter 11 of the Rustonomicon[14] reads,

> It's important to be mindful of `panic!`s when working with FFI. A `panic!` across an FFI boundary is undefined behavior. If you're writing code that may panic, you should run it in a closure with `catch_unwind`.

This is, however, not quite a perfect solution, as the documentation for `catch_unwind` explains (emphasis in original):

> Note that this function **may not catch all panics** in Rust. A panic in Rust is not always implemented via unwinding, but can be implemented by aborting the process as well. This function *only* catches unwinding panics, not those that abort the process.[15]

The impact of this issue is impossible to determine with certainty, since undefined behavior is by definition unpredictable; however, it should still be taken seriously. Glancing at the historical record of undefined-behavior-related bugs, it may be observed that assumptions about what undefined behavior might (or might not) lead to are almost always mistaken (and given the degree of transformation performed by modern optimizing compilers, this is truer now than ever). This exposes calling code to a level of risk that is uncharacteristic for a Rust library and inappropriate for sensitive applications.

**Recommendation**

Ensure that unwinding panics in the FFI are handled with `catch_unwind` (or removed), and functions which would otherwise panic are rewritten to instead return caller-legible error codes. Ensure that code paths which could trigger aborting panics are avoided altogether.

**Retest Results**

As part of Pull Request 48, the Horizen Labs team made a conscious effort to fix instances of crash-inducing constructions, for example by converting many `unwrap()` calls to more explicit error handling. Additionally, the team added the following directive to the `Cargo.toml` file:

```
[profile.release]
panic = 'abort'
```

The outcome of that change is that program executions will immediately abort upon panics,

[14] See Rustonomicon Chapter 11, subheading "FFI and panics"
[15] See the Rust docs for std::panic::catch_unwind

and as such panics will not *cross* FFI boundaries. As a result, this finding was marked as "Fixed".

The NCC Group team noted that this change may have unintended consequences, since processes dying abruptly do not get a chance to clean up anything. Specifically, destructors of locally allocated objects may not get called, temporary files may not be deleted and data may be lost (for example if some process had written data to a file but the data was still held in a buffer in the process address space because the process did not call `fflush()`).

**Finding** **Non Constant-Time Modular Exponentiation**

**Risk** **Informational** Impact: Low, Exploitability: Low

**Identifier** NCC-E001741-003

**Status** Risk Accepted

**Category** Cryptography

**Component** ginger-lib

**Location** `algebra/src/fields/mod.rs`

**Impact** An adversary may be able to infer the value of the exponent through side-channel leaks. In case the exponent is secret, this may constitute an important confidentiality breach.

**Description** Modular exponentiation of field elements is performed by the `pow()` function located in `algebra/src/fields/mod.rs`, and provided below for reference. This function implements a simple *binary exponentiation* algorithm,[16] which branches conditionally based on the current bit value of the exponent being iterated over, as can be seen in the highlighted code portion below.

```
fn pow<S: AsRef<[u64]>>(&self, exp: S) -> Self {
    let mut res = Self::one();

    let mut found_one = false;

    for i in BitIterator::new(exp) {
        if !found_one {
            if i {
                found_one = true;
            } else {
                continue;
            }
        }

        res.square_in_place();

        if i {
            res *= self;
        }
    }
    res
}
```

This conditional branch will incur different computational load based on the exponent value. Under certain conditions, this timing leak may be observed by an attacker and used to recover the exponent.

**Recommendation** Consider writing a constant-time modular exponentiation function, namely, a function that performs the same amount of computation regardless of its input.

BearSSL[17] and the GitHub Cryptocoding[18] repository have valuable documentation about side-channel attacks and how to avoid them.

[16] https://en.wikipedia.org/wiki/Modular_exponentiation#Right-to-left_binary_method
[17] https://www.bearssl.org/constanttime.html
[18] https://github.com/veorq/cryptocoding

**Retest Results** With Pull Request 112, the following disclaimer was added to the different `mul_assign()`, `mul_bits()` and `pow()` functions:

```
    /// WARNING: This implementation doesn't take costant time with respect
    /// to the exponent, and therefore is susceptible to side-channel attacks.
    /// Be sure to use it in applications where timing (or similar) attacks
    /// are not possible.
    /// TODO: Add a side-channel secure variant.
```

This finding was marked as “Risk Accepted” as a result.

| Finding | **Missing Memory Zeroization** |
|---|---|
| Risk | **Informational** Impact: Low, Exploitability: Low |
| Identifier | NCC-E001741-007 |
| Status | Risk Accepted |
| Category | Data Exposure |
| Component | Systemic |
| Location | Systemic |
| 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 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.<br><br>The different repositories in scope do not exhibit particular care for memory zeroization; in no instance were they observed to erase sensitive data. For example, no steps are taken to ensure the random mask polynomials (used to achieve zero-knowledge by masking the polynomials `w_poly`, `z_a_poly` and `z_b_poly`, and previously discussed in another context in finding NCC-E001741-017 on page 11), are being correctly zeroized .<br><br>As another example, although outside the scope of the review, the NCC Group team noted that the `ginger-lib` library was not performing memory zeroization for secret keys, for example in the Schnorr-based signature `SecretKey` structure used in `primitives/src/signature/schnorr/field_based_schnorr.rs`.<br><br>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"[19] 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 Zeroize[20] crate. |
| Recommendation | Utilize the Zeroize crate to derive the zeroize-on-drop trait for all sensitive values.<br><br>Ensure the same approach is taken to attach the zeroize-on-drop trait to all secret material found in the Rust bindings. |
| Retest Results | The Horizen Labs team indicated that this finding would be addressed at a later stage. In the meantime, an issue was opened on Github to track the status of memory zeroization. As a result, this finding was marked as "Risk Accepted". |

[19] https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_zhaomo_yang.pdf
[20] https://docs.rs/zeroize/1.1.1/zeroize/

| | |
|---|---|
| **Finding** | **Potential to Randomly Generate Trivial Random Challenges** |
| **Risk** | **Informational** Impact: Medium, Exploitability: None |
| **Identifier** | NCC-E001741-020 |
| **Status** | Fixed |
| **Category** | Cryptography |
| **Component** | Systemic |
| **Location** | • `poly-commit/src/ipa_pc/mod.rs`<br>• `poly-commit/src/lib.rs`<br>• `marlin/src/ahp/verifier.rs`<br>• `ginger-lib/proof-systems/src/darlin/accumulators/dlog.rs` |
| **Impact** | Zero field elements may be generated as random challenges, potentially resulting in unexpected behavior or Rust panics. |

**Description**

While both Darlin[21] and Marlin[22] make extensive use of random field elements as challenges, the reference academic papers do not impose restrictions about the fact that they should be non-zero.

However, the NCC Group team noted that the generation of the zero field element by the Random Number Generator (RNG) would result in unexpected behavior and panics.

For example, in the function `open_check_polys()` in `src/ipa_pc/mod.rs`, the `inverse()` function call is performed on the freshly-generated challenge, after which a call to `unwrap()` will panic in case the `round_challenge` is zero (since zero does not admit an inverse in a finite field). This also happens in the functions `open_individual_opening_challenges()` and `succinct_check()` of the same file.

```
round_challenge = fs_rng.squeeze_128_bits_challenge();

let round_challenge_inv = round_challenge.inverse().unwrap();
```

Note that the function `squeeze_128_bits_challenge()` in `poly-commit/src/rng.rs` does not check that the resulting element is non-zero.

```
/// Squeeze a new random field element
fn squeeze_128_bits_challenge<F: Field>(&mut self) -> F {
    u128::rand(self).into()
}
```

Similarly, in the function `succinct_batch_check_individual_opening_challenges()` in `src/ipa_pc/mod.rs`, the generation of zero for the `lambda` parameter would lead to `cur_challenge` being zero. Additionally, if the random challenge `point` was equal to any of the `x_i`'s in the code excerpt below, a zero-division would be triggered on the penultimate highlighted line:

```
// lambda
let lambda: G::ScalarField = fs_rng.squeeze_128_bits_challenge();
let mut cur_challenge = G::ScalarField::one();
```

[21] *Darlin: Recursive Proofs using Marlin* https://eprint.iacr.org/2021/930
[22] *Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS* https://eprint.iacr.org/2019/1047

```
// Fresh random challenge x
fs_rng.absorb(&to_bytes![batch_commitment].unwrap());
let point: G::ScalarField = fs_rng.squeeze_128_bits_challenge();

let mut computed_batch_v = G::ScalarField::zero();

for ((&v_i, y_i), x_i) in v_values.iter().zip(y_values).zip(points) {

    computed_batch_v = computed_batch_v + &(cur_challenge * &((v_i - &y_i)
     → / &(point - x_i)));

    cur_challenge = cur_challenge * &lambda;
}
```

Note that other instances exist throughout the different code bases, where the generation of the zero field element could result in the zero-knowledge property of some protocols to not be fulfilled.

However, in the absence of other implementation issues, and provided that the underlying RNG is secure, the probability of generating the zero field element at random is negligible.

**Recommendation** Consider going through the code base to identify areas where the generation of the zero element would result in insecure computations or panics. Pay particular attention to areas where possible adversarial input is used together with randomly generated elements (for example in potential zero-division cases, as described above), since adversaries may be able to trigger unexpected edge cases.

Additionally, consider updating the reference papers and implementations to sample the random challenges from $\mathbb{F}^*$ (and not from $\mathbb{F}$) where appropriate.

**Retest Results** With Pull Request 20, the `squeeze_128_bits_challenge()` function was updated to prevent sampling zero, as follows:

```
self.gen_range(1u128..u128::MAX).into()
```

This addresses the issue outlined above and this finding has been marked as "Fixed" as a result. Note however that the range defined above excludes the upper bound. In order to define an inclusive range, the following line could be used:

```
self.gen_range(1u128..=u128::MAX).into()
```

# Appendix A: 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.

**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.

**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.

**High** Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

**Medium** 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.

| | |
|---|---|
| **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. |

This informational section highlights a number of observations that do not warrant security-related findings on their own.

### ginger-lib

- The following Montgomery reduction routine in `algebra/src/fields/arithmetic.rs` is repeated at the end of three macro implementations, namely in `mul_assign()`, `into_repr()` and `square_in_place()`.

```
  // Montgomery reduction
let mut _carry2 = 0;
for i in 0..$limbs {
    let k = r[i].wrapping_mul(P::INV);
    let mut carry = 0;
    fa::mac_with_carry(r[i], k, P::MODULUS.0[0], &mut carry);
    for j in 1..$limbs {
        r[j + i] = fa::mac_with_carry(r[j + i], k, P::MODULUS.0[j], &mut carry);
    }
    r[$limbs + i] = fa::adc(r[$limbs + i], _carry2, &mut carry);
    _carry2 = carry;
}
(self.0).0.copy_from_slice(&r[$limbs..]);
self.reduce();
```

  Consider moving this code into its own function.

- The `legendre()` function in `algebra/src/fields/macros.rs` could be slightly optimized to not perform the `pow` if `self` was zero.

```
fn legendre(&self) -> LegendreSymbol {
    use crate::fields::LegendreSymbol::*;

    // s = self^((MODULUS - 1) // 2)
    let s = self.pow(P::MODULUS_MINUS_ONE_DIV_TWO);
    if s.is_zero() {
        Zero
    } else if s.is_one() {
        QuadraticResidue
    } else {
        QuadraticNonResidue
    }
}
```

- Some comments in the Tweedle Curve parameters source files (`algebra/src/curves/tweedle/dee.rs` and `algebra/src/curves/tweedle/dum.rs`) are slightly misleading. Specifically, the comments describe the values of some constants in their "normal" forms, while their actual values are in Montgomery representation.

```
/// COEFF_B = 5
const COEFF_B: Fq = field_new!(
    Fq,
    BigInteger256([
        0x30aef343ffffffed,
        0xbcb60a132dafff0b,
        0xffffffffffffffff,
        0x3fffffffffffffff
    ])
);
```

```
/// COFACTOR = 1
const COFACTOR: &'static [u64] = &[0x1];

/// COFACTOR_INV = 1
const COFACTOR_INV: Fr = field_new!(
    Fr,
    BigInteger256([
        0x1c3ed159fffffffd,
        0xf5601c89bb41f2d3,
        0xffffffffffffffff,
        0x3fffffffffffffff
    ])
);
```

- The declaration of the `buckets` vector for multi-scalar multiplication in the file `algebra/src/msm/variable_base.rs` initializes the size of buckets to `bases.len()/cc * 2`. Is this size optimal?

```
let mut buckets = vec![Vec::with_capacity(bases.len()/cc * 2); cc];
```

- The function `reindex_by_subdomain()` in `algebra/src/fft/domain/mod.rs` does not seem to be used anywhere. It presents a few opportunities for panics, like in the division highlighted below. Consider performing stricter parameter validation and removing all unused functions.

```
  /// Given an index which assumes the first elements of this domain are the elements of
/// another (sub)domain with size size_s, this returns the actual index into this domain.
fn reindex_by_subdomain(&self, other_size: usize, index: usize) -> usize {
    assert!(self.size() >= other_size);
    // Let this subgroup be G, and the subgroup we're re-indexing by be S.
    // Since its a subgroup, the 0th element of S is at index 0 in G, the first element of S is at
    // index |G|/|S|, the second at 2*|G|/|S|, etc.
    // Thus for an index i that corresponds to S, the index in G is i*|G|/|S|
    let period = self.size() / other_size;
    if index < other_size {
        index * period
    } else {
        // ...
    }
```

- Some functions in `algebra/src/fft/polynomial/dense.rs` seem to have copy-pasted comments that do not apply, such as in the `mul_by_vanishing_poly()` function where a comment describes a division operation.

```
/// Multiply `self` by the vanishing polynomial for the domain `domain`.
/// Returns the quotient and remainder of the division.
pub fn mul_by_vanishing_poly(&self, domain_size: usize) -> DensePolynomial<F> {
```

- There are outdated comments in `algebra/src/fft/polynomial/sparse.rs`. Since these functionalities are now implemented, they should probably be deleted.

```
// unimplemented!("current implementation does not produce evals in correct order")
```

- In `primitives/src/merkle_tree/mod.rs`, the highlighted code block below seems superfluous, since an error is triggered a few lines above if the `path` length is not equal to `P::HEIGHT`.

```
 if self.path.len() != P::HEIGHT as usize {
            return Err(MerkleTreeError::IncorrectPathLength(self.path.len(), P::HEIGHT as usize))?
        }
        // Check that the given leaf matches the leaf in the membership proof.
        let mut buffer = vec![0u8; P::H::INPUT_SIZE_BITS/8];

        if !self.path.is_empty() {
        // ...
```

- In `primitives/src/merkle_tree/mod.rs`, it is unclear why the functions `hash_leaf()` and `hash_inner_node()` take a buffer as argument. Consider the function `hash_leaf()` provided below for reference. The buffer is only used temporarily to write the content of the leaf.

```
/// Returns the hash of a leaf.
pub(crate) fn hash_leaf<H: FixedLengthCRH, L: ToBytes>(
    parameters: &H::Parameters,
    leaf: &L,
    buffer: &mut [u8],
) -> Result<H::Output, Error> {
    use std::io::Cursor;
    let mut writer = Cursor::new(buffer);
    leaf.write(&mut writer)?;

    let buffer = writer.into_inner();
    H::evaluate(parameters, &buffer[..])
}
```

In comparison, the function `hash_empty()` in the same file only declares a local buffer and passes it to the `H::evaluate` call.

```
  pub(crate) fn hash_empty<H: FixedLengthCRH>(
    parameters: &H::Parameters,
) -> Result<H::Output, Error> {
    let empty_buffer = vec![0u8; H::INPUT_SIZE_BITS / 8];
    H::evaluate(parameters, &empty_buffer)
}
```

- The file `primitives/src/crh/poseidon/parameters/tweedle_dee.rs` has a number of misleading and outdated comments that should be deleted.

```
  // Number of partial rounds
  const R: usize = 2;  // The rate of the hash function

  // ...

  // For rounds 4 + 56 + 4 = 65
```

- In the same file, it is slightly confusing that the `tweedle::Fq` field is being renamed to `Fr`: `use algebra::fields::tweedle::Fq as Fr;`. Specifically, the Tweedle Dee and Dum Poseidon instances *seem* to be using the same base field, while they're *actually only* using the same notation but for different prime fields.

- In the file `primitives/src/merkle_tree/field_based_mht/optimized/mod.rs`, the `init()` function initializes an optimized Merkle Tree. It populates some vectors of indices, but in the highlighted code it should probably be `T::MERKLE_ARITY` instead of `rate` (though an assertion currently ensures they are the same).

```
    // Compute indexes
    while size >= 1 {
        initial_pos.push(initial_idx);
        final_pos.push(final_idx);
        processed_pos.push(initial_idx);
        new_elem_pos.push(initial_idx);

        initial_idx += size;
        size /= rate;
        final_idx = initial_idx + size;
    }
```

### Marlin

- In the file `src/ahp/verifier.rs`, the function `verifier_first_round()` performs a polynomial evaluation at a random point as part of the verification procedure, and triggers a panic if the result is zero.

```
let alpha: F = fs_rng.squeeze_128_bits_challenge();
assert!(!domain_h.evaluate_vanishing_polynomial(alpha).is_zero());
```

The function `verifier_second_round()` in the same file performs a similar computation.

```
let beta: F = fs_rng.squeeze_128_bits_challenge();
assert!(!state.domain_h.evaluate_vanishing_polynomial(beta).is_zero());
```

Consider returning a verification error instead of triggering a panic if the polynomials evaluate to `0`.

### zendoo-cctp-lib

- In `src/proving_system/mod.rs` the serialization identifiers for Marlin and Darlin are hardcoded and repeated, for example in `serialize()`

```
match self {
    ProvingSystem::Undefined => CanonicalSerialize::serialize(&0u8, writer),
    ProvingSystem::Darlin => CanonicalSerialize::serialize(&1u8, writer),
    ProvingSystem::CoboundaryMarlin => CanonicalSerialize::serialize(&2u8, writer)
}
```

and in `deserialize()`

```
0u8 => Ok(ProvingSystem::Undefined),
1u8 => Ok(ProvingSystem::Darlin),
2u8 => Ok(ProvingSystem::CoboundaryMarlin),
```

among others. Consider defining and using symbolic constants for these values.

### zendoo-mc-cryptolib

- There are a few commented `println` statements in `src/macros.rs`

```
if buffer.is_null() {
    //println!("===> ERR CODE {:?}", CctpErrorCode::NullPtr);
    return (false, CctpErrorCode::NullPtr)
}
```

- In `src/type_mapping.rs`, consider setting `UINT_160_SIZE` to `20` explicitly, since this value is constant, while `MC_P`

`K_SIZE` might not be.

```
pub const UINT_160_SIZE: usize = MC_PK_SIZE; //in bytes
```