Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

ChainSafe

BLS12-381 Implementation

Veridise Inc.
May 15, 2024

► **Prepared For:**

ChainSafe
`https://chainsafe.io/`

► **Prepared By:**

Jon Stephens
Benjamin Mariano
Alp Bassa

► **Contact Us:** `contact@veridise.com`

► **Version History:**

May. 14 2024 V1

Contents

1 Executive Summary

From Apr. 8, 2024 to Apr. 18, 2024, ChainSafe engaged Veridise to review the security of their BLS12-381 Implementation. The review covered their Rust implementation of BLS12-381 elliptic-curve-based signatures, which enables writing zero-knowledge proof circuits including BLS12-381 curve operations using the Halo2 proving stack. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks on commit `723962a`. The auditing strategy involved extensive manual analysis of the source code performed by Veridise engineers.

Project summary. The security assessment covered the addition of BLS12-381 curve operations to the existing `halo2-lib` library. The additions are intended to allow developers to validate BLS12-381 operations in a ZK circuit. They therefore add implementations for several chips that contain behaviors similar to the existing BN254 curve. These include the ability to check pairings, validate BLS signatures, create G1 and G2 subgroup points, and perform various operations on the curve.

In addition, the review covered the addition of a generic implementation of the hash-to-curve protocol which takes an arbitrary message and maps it to a point on the curve. At a high level, this is done by first constructing the cryptographic hash of the message and then mapping the resulting hash to a point on the curve. The added functionality implements the hash-to-curve protocol and also allows developers to customize the cryptographic hash function and curve used by providing chips that implement the required functionality. In addition to this, the developers also provide the necessary functionality for the BLS12-381 curve.

Code assessment. The BLS12-381 Implementation developers provided the source code of the BLS12-381 Implementation contracts for review. The source code appears to be mostly original code written by the BLS12-381 Implementation developers. However, the code is heavily inspired by the existing implementation of the BN254 elliptic curve as well as the Hashing To Elliptic Curve RFC. The code contains some documentation in the form of READMEs, documentation comments on functions and storage variables, and explanation of ZK constraints.

The source code contained a test suite, which the Veridise auditors noted tested most of the high-level expected behaviors for constructing signatures using BLS12-381. However, it should be noted that at the time of the audit, not all tests pass. The failing tests appear to be for benchmarking purposes and the developers noted that they are not necessarily intended to pass.

Summary of issues detected. The audit uncovered 17 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-CSB-VUL-001 identifies unconstrained values used when verifying BLS signatures and V-CSB-VUL-002 corresponds to an unconstrained constant that is used when computing `cyclomic_pow`. The Veridise auditors also identified 4 medium-severity issues, including a logical error that could produce malformed big integers from bytes (V-CSB-VUL-003) and a missing zero check that could allow a division to return an unconstrained value (V-CSB-VUL-006). Additionally, 6 low-severity issues, 4 warnings, and 1 informational findings were also reported. The developers fixed all of the issues reported as high or medium, and a majority of the low, warning and informational issues as well.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the new BLS12-381 and hash-to-curve implementations.

Improve Test Quality. While the project was delivered with tests that appear to exercise most high-level functionality, we believe the quality could be improved by increasing test coverage, particularly for some of the more complicated helper functions and lower-level APIs.

Document Assumptions. A few locations do not document important assumptions being made by a function. In particular, V-CSB-VUL-010, V-CSB-VUL-014, and V-CSB-VUL-016 identified cases where functions were correct if used in some expected context, but may not be properly constrained if provided with untrusted input. We would recommend that these assumptions be documented so that future developers are aware of additional constraints they may need to provide.

Use Rust Macros. As discussed in issue V-CSB-VUL-013, we would recommend that the developers make use of macros such as `#[must_use]`. Such macros will throw warnings at compile time if an API is not used as intended, which can enhance the quality of projects that use the library.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
BLS12-381 Implementation	`723962a`	Rust	Halo2

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Apr. 8 - Apr. 18, 2024	Manual	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	2	2	2
Medium-Severity Issues	4	2	2
Low-Severity Issues	6	6	5
Warning-Severity Issues	4	4	4
Informational-Severity Issues	1	1	1
TOTAL	17	15	14

Table 2.4: Category Breakdown.

Name	Number
Under-constrained Cell	5
Logic Error	4
Data Validation	4
Maintainability	4

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the BLS12-381 Implementation in `halo2-lib`. In our audit, we sought to answer questions such as:

- Do hash-to-curve operations match the algorithms described in Hashing To Elliptic Curve RFC?
- Does the BLS12-381 implementation match the BN254 implementation (as is appropriate)?
- Are all constraints appropriately constructed to avoid under/over-constrained issues?
- Are there potential vulnerabilities due to over/underflow of Rust integers?
- Can an attacker get a signature verified as matching a public key when it in fact does not?
- Can an attacker perform a rogue public-key attack?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved intense scrutiny of the code by human experts. The audit focused on validating the correctness of the BLS12-381 curve operations with respect to existing implementations as well as verification that constraints are correctly provided to avoid attacks. In particular, we conducted the audit with the aid of the following techniques:

- *Property-based Testing.* We leveraged property-based testing to determine if the protocol may deviate from the expected behavior. To do this, we identified several properties that should always hold and wrote tests to pseudo-randomly exercise the protocol and check these properties.
- *Differential Fuzzing.* To check for behaviors inconsistent with existing implementations of the BLS12-381 curve, we performed differential fuzzing. To do so, we identified an appropriate reference implementation (the `bls12_381` crate) and executed both the new and reference implementation on pseudo-random inputs then compared the results to ensure there were no deviations.

Scope. The scope of this audit is limited to the changes in this pull request restricted to the following directories:

- `halo2-ecc/src/bls12_381`
- `halo2-ecc/src/ecc`

Methodology. Veridise auditors reviewed security reports for similar elliptic curve audits, inspected the provided tests, and read the BLS12-381 Implementation documentation (as well as external documentation around safe implementation of BLS12-381). During the audit, the Veridise auditors communicated with the ChainSafe developers over Telegram as necessary to raise issues and clarify questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Vulnerability Report 4

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-CSB-VUL-001	Errors in bls_signature_verify	High	Fixed
V-CSB-VUL-002	Unconstrained constant	High	Fixed
V-CSB-VUL-003	Error in conversion from Bytes to BigInt	Medium	Fixed
V-CSB-VUL-004	Deviating representation order for FP12 elements	Medium	Intended Behavior
V-CSB-VUL-005	Wrong implementation of line equation formula	Medium	Intended Behavior
V-CSB-VUL-006	Unsafe division by zero	Medium	Fixed
V-CSB-VUL-007	No API to check membership in sub-group	Low	Acknowledged
V-CSB-VUL-008	Under-constrained is_square flag in sqrt_ratio	Low	Fixed
V-CSB-VUL-009	hash_to_field msg could be unconstrained	Low	Fixed
V-CSB-VUL-010	Unchecked hash length assumption	Low	Fixed
V-CSB-VUL-011	Missing check for expand_message	Low	Fixed
V-CSB-VUL-012	Missing equality check	Low	Fixed
V-CSB-VUL-013	Consider adding must_use macro	Warning	Fixed
V-CSB-VUL-014	User must determine sign of ratio root	Warning	Fixed
V-CSB-VUL-015	Misleading, outdated, missing or wrong comments	Warning	Fixed
V-CSB-VUL-016	Undocumented non-zero input assumption	Warning	Fixed
V-CSB-VUL-017	Unnecessary clones	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-CSB-VUL-001: Errors in bls_signature_verify

Severity	High	**Commit**	0350dd7
Type	Logic Error	**Status**	Fixed
File(s)	`bls12_381/bls_signature.rs`		
Location(s)	bls_signature_verify()		
Confirmed Fix At	6a9c9ea		

The `bls_signature_verify` function, shown below, is intended to verify BLS signatures on the BLS12-381 curve. This function, however, does not validate some important pieces of information. In particular, it does not validate that any of the inputs are on the curve or that they are in the appropriate subgroup. Additionally, since `load_private_g2_unchecked` and `load_private_g2_unchecked` simply create new unconstrained witness cells, the inputs are unconstrained.

```
pub fn bls_signature_verify(
    &self,
    ctx: &mut Context<F>,
    signature: G2Affine,
    pubkey: G1Affine,
    msghash: G2Affine,
) {
    let signature_assigned = self.pairing_chip.load_private_g2_unchecked(ctx,
    signature);
    let pubkey_assigned = self.pairing_chip.load_private_g1_unchecked(ctx, pubkey);
    let hash_m_assigned = self.pairing_chip.load_private_g2_unchecked(ctx, msghash);

    self.assert_valid_signature(ctx, signature_assigned, hash_m_assigned,
    pubkey_assigned);
}
```

Snippet 4.1: Snippet from `bls_signature_verify`

Impact The missing validation could cause the function to be susceptible to well-known attacks on bls signature schemes such as small subgroup attacks. In addition, since the validation is performed on unconstrained input signals which are not returned, it will be difficult for the user to add the missing validation.

Recommendation Consider returning `signature_assigned`, `pubkey_assigned` and `hash_m_assigned` and adding the missing validation. Alternatively, consider making the function only available to tests.

4.1.2 V-CSB-VUL-002: Unconstrained constant

Severity	High	**Commit**	0350dd7
Type	Under-constrained Cell	**Status**	Fixed
File(s)	`bls12_381/final_exp.rs`		
Location(s)	cyclotomic_pow()		
Confirmed Fix At	36c4e5b		

In Axiom's Halo2 implementation, users have multiple methods to allocate and initialize a cell in the underlying table. The `cyclomatic_pow` function, which is shown below, calls the `load_private` function. This function then allocates and returns a vector of big integers which reference cells in the plonk table. The referenced cells in the table are not constrained, except for having the necessary range checks to verify the value is well-structured. As such, this value can signify any integer of the correct size and is anticipated to be further constrained once returned. It's crucial to note that while this API does accept a big integer value as an input, this input is the value assigned during the witness generation and doesn't constrain the result.

```rust
pub fn cyclotomic_pow(&self, ctx: &mut Context<F>, a: FqPoint<F>, exp: u64) ->
    FqPoint<F> {
    let mut res = self.load_private(ctx, Fq12::one());
    let mut found_one = false;

    for bit in (0..64).rev().map(|i| ((exp >> i) & 1) == 1) {
        if found_one {
            let compressed = self.cyclotomic_square(ctx, &self.cyclotomic_compress(&
    res));
            res = self.cyclotomic_decompress(ctx, compressed);
        } else {
            found_one = bit;
        }

        if bit {
            res = self.mul(ctx, &res, &a);
        }
    }

    self.conjugate(ctx, res)
}
```

Snippet 4.2: Definition of the `cyclomatic_pow` function

Impact It is intended for the initial `res` value of the `cyclomatic_pow` function to be `Fq12::one()` however `load_private` will not constrain `res` so that it may only refer to this value. The initial `res` value is therefore under-constrained and as all of the remaining computation uses this value without further constraining it, so to will be the return value of `cyclomatic_pow`.

Recommendation Rather than invoking `load_private`, instead call `load_constant` as this function will constrain the resulting value to always be `Fq12::one()`

4.1.3 V-CSB-VUL-003: Error in conversion from Bytes to BigInt

Severity	Medium	**Commit**	0350dd7
Type	Logic Error	**Status**	Fixed
File(s)	`bigint/utils.rs`		
Location(s)	decode_into_bn()		
Confirmed Fix At	ae6fcfd		

The `decode_into_bn` function will decode a list of bytes into a big integer. In the process, it attempts to determine if the input bytes are already in an appropriate format and can be directly used as big integer limbs. It will currently do so in two cases, if the integer is intended to represent a bit and in the case where each limb in the resulting big integer is 64 bits. The second case, however, will not create a valid big integer as each value in `bytes` is 8 bits.

```
pub fn decode_into_bn<F: BigPrimeField>(
    ctx: &mut Context<F>,
    gate: &impl GateInstructions<F>,
    bytes: Vec<AssignedValue<F>>,
    limb_bases: &[F],
    limb_bits: usize,
) -> ProperCrtUint<F> {
    let limb_bytes = limb_bits / 8;
    let bits = limb_bases.len() * limb_bits;

    ...

    // inputs is a bool or uint8.
    let assigned_uint = if bits == 1 || limb_bytes == 8 {
        ProperUint(bytes)
    } else {
        ...
    };

    assigned_uint.into_crt(ctx, gate, value, limb_bases, limb_bits)
}
```

Snippet 4.3: Definition of the `decode_into_bn` function

Impact In the case where `limb_bits` is 64, the resulting integer will be incorrect.

Recommendation Remove the `limb_bytes == 8` condition. Instead, consider checking `limb_bits == 8`, `limb_bytes == 1` or `bits == 8`

4.1.4 V-CSB-VUL-004: Deviating representation order for FP12 elements

Severity	Medium	**Commit**	0350dd7
Type	Logic Error	**Status**	Intended Behavior
File(s)	`bls12_381/pairing.rs`		
Location(s)	sparse_line_function_unequal(), sparse_line_function_equal()		
Confirmed Fix At			

The functions `sparse_line_function_unequal` and `sparse_line_function_equal` return elements of $\mathbb{F}_{p^{12}}$ represented by a 6-tuple $(b_0, b_2, \ldots, b_5)$ of elements of $\mathbb{F}_{p^2}$ to represent $b_0+b_1\cdot w+\cdots+b_5\cdot w^5$, where $\mathbb{F}_{q^{12}} = \mathbb{F}_{q^2}(w)$ with $w^6 = u + 1$.

For the function `sparse_line_function_unequal` the expression has w^2, w^3 and w^5 terms, however the output is given as `[None, Some(out2), None, Some(out3), Some(out4), None]` where `out2`, `out3` and `out4` are the coefficients of w^5, w^3 and w^2 respectively, which does not agree with the above representation.

Similarly, for `sparse_line_function_equal` the output if given as `[Some(out0), None, Some(out2 ), Some(out3), None, None]` whereas there are terms with w^0, w^3 and w^4. The ordering of the elements in the output vector needs to be reconsidered.

Impact Incorrect ordering in the representation could lead to unexpected behavior.

Recommendation Correct the output orders.

Update Some of the comments on the function's behavior were inaccurate which misled the auditors. The comments have been updated to be consistent with the behavior of the function.

4.1.5 V-CSB-VUL-005: Wrong implementation of line equation formula

Severity	Medium	**Commit**	0350dd7
Type	Logic Error	**Status**	Intended Behavior
File(s)	`bls12_381/pairing.rs`		
Location(s)	sparse_line_function_equal()		
Confirmed Fix At			

In the function `sparse_line_function_equal`, the expression for the line function in the case of the tangent is given by:

$(3x^3 - 2y^2) \cdot w^6 + (-3x^2 \cdot Q.x) \cdot w^4 + (2y \cdot Q.y) \cdot w^3$

The tower finite fields is defined as

$\mathbb{F}_{q^2} = \mathbb{F}_q(u)$ with $u^2 = -1$,

$\mathbb{F}_{q^{12}} = \mathbb{F}_{q^2}(w)$ with $w^6 = u + 1$.

The w^6 in the first term above gives a $u + 1$ factor, which is missing in the implementation of `sparse_line_function_equal`. As such, an additional call to `mul_no_carry_w6::<_, _, 1>` must be added.

Impact Using a wrong expression for the line will cause unintended behavior.

Recommendation Add a factor of $u+1$ to the corresponding term by a call to `mul_no_carry_w6` .

Update Some of the comments on the function's behavior were inaccurate which misled the auditors. The comments have been updated to be consistent with the behavior of the function.

4.1.6 V-CSB-VUL-006: Unsafe division by zero

Severity	Medium	**Commit**	0350dd7
Type	Under-constrained Cell	**Status**	Fixed
File(s)	`bls12_381/hash_to_curve.rs`		
Location(s)	isogeny_map()		
Confirmed Fix At	fc8d2ce		

The `isogeny_map` function is intended to take a point in the field and map it to a point on the curve. The implementation of this function is given below.

```rust
fn isogeny_map(&self, ctx: &mut Context<F>, p: G2Point<F>) -> G2Point<F> {
    let fp2_chip = self.field_chip();
    // constants
    let iso_coeffs = [
        G2::ISO_XNUM.to_vec(),
        G2::ISO_XDEN.to_vec(),
        G2::ISO_YNUM.to_vec(),
        G2::ISO_YDEN.to_vec(),
    ]
    .map(|coeffs| coeffs.into_iter().map(|iso| fp2_chip.load_constant(ctx, iso)).
    collect_vec());

    let fq2_zero = fp2_chip.load_constant(ctx, Fq2::ZERO);

    let [x_num, x_den, y_num, y_den] = iso_coeffs.map(|coeffs| {
        coeffs.into_iter().fold(fq2_zero.clone(), |acc, v| {
            let acc = fp2_chip.mul(ctx, acc, &p.x);
            let no_carry = fp2_chip.add_no_carry(ctx, acc, v);
            fp2_chip.carry_mod(ctx, no_carry)
        })
    });

    let x = { fp2_chip.divide_unsafe(ctx, x_num, x_den) };

        let y = {
        let tv = fp2_chip.divide_unsafe(ctx, y_num, y_den);
        fp2_chip.mul(ctx, &p.y, tv)
    };

    G2Point::new(x, y)
}
```

Snippet 4.4: Implementation of `isogeny_map`

The line `let x = { fp2_chip.divide_unsafe(ctx, x_num, x_den) };` uses the `divide_unsafe` function, which leads to potentially unconstrained values when `x_den` is `0`. Constraints are never added to assert that `x_den` is non-zero. The same concern exists for the line `let tv = fp2_chip.divide_unsafe(ctx, y_num, y_den);`.

Impact In the case that `x_den` and/or `y_den` are `0`, the points on the curve `x,y` returned by the function are potentially under-constrained.

In the expected use case where the input to the function is the result of a hash function, the likelihood of `x_den` or `y_den` being `0` is quite low (and is definitionally very difficult for an attacker to intentionally choose). However, this function is public, meaning there is no guarantee the function is used with the input from a hash function as intended.

Recommendation Add checks that the denominators are non-zero.

4.1.7 V-CSB-VUL-007: No API to check membership in sub-group

Severity	Low	**Commit**	0350dd7
Type	Data Validation	**Status**	Acknowledged
File(s)	`pairing.rs`		
Location(s)	N/A		
Confirmed Fix At	N/A		

To avoid some common attacks such as small sub-group attacks, it can be important to check that a point is in the proper subgroup. No such function exists despite there being an API in BLS12-381 to create a G1 and G2 point.

Recommendation We would recommend adding a function for those building on this library in the future.

4.1.8 V-CSB-VUL-008: Under-constrained is_square flag in sqrt_ratio

Severity	Low	**Commit**	0350dd7
Type	Under-constrained Cell	**Status**	Fixed
File(s)	ecc/hash_to_curve.rs		
Location(s)	sqrt_ratio()		
Confirmed Fix At	35b54cd		

The `sqrt_ratio` function encodes the following behavior:

```
is_square = True and y = sqrt(num / div) if (num / div) is square in F, and
is_square = False and y = sqrt(Z * (num / div)) otherwise.
```

Snippet 4.5: Specification for `sqrt_ratio`

The `sqrt_ratio` function does so by first allocating the `is_square` and `y_assigned` values in the plonk table. It then computes the values of `num / div` as `ratio` and `Z * num / div` as ratio_z. The `y_check` signal is then constrained to be the value selected by `is_square` and to be equal to the square of the return value. The values `y_assigned` and `is_square` are therefore constrained as long as `ratio` and `ratio_z` are not both squares. Since `Z` itself is not square, only one of `ratio` and `ratio_z` may be square as long as the input `num` is not zero. This function does not check that the input value is non-zero, however, and as a result `is_square` may only be constrained to be a bit.

```
fn sqrt_ratio(
    &self,
    ctx: &mut Context<F>,
    num: FC::FieldPoint,
    div: FC::FieldPoint,
) -> (AssignedValue<F>, FC::FieldPoint) {
    ...

    let is_square = ctx.load_witness(F::from(is_square.unwrap_u8() as u64));
    field_chip.gate().assert_bit(ctx, is_square); // assert is_square is boolean

    let y_assigned = field_chip.load_private(ctx, y);
    let y_sqr = field_chip.mul(ctx, y_assigned.clone(), y_assigned.clone()); // y_sqr
     = y1^2

    let ratio = field_chip.divide(ctx, num, div); // r = u / v

    let swu_z = field_chip.load_constant(ctx, C::SWU_Z);
    let ratio_z = field_chip.mul(ctx, ratio.clone(), swu_z.clone()); // r_z = r * z

    let y_check = field_chip.select(ctx, ratio, ratio_z, is_square); // y_check =
    is_square ? ratio : r_z

    field_chip.assert_equal(ctx, y_check, y_sqr); // assert y_check == y_sqr

    (is_square, y_assigned)
}
```

Snippet 4.6: Definition of the `sqrt_ratio` function

Impact If `num` is zero, the resulting `y_assigned` value will be zero and `is_square` can be assigned to 0 or 1. Since the `is_square` signal is often used to select values, this could result in an incorrect signal being selected. It should be noted however that it may be difficult for an attacker to control `num` as `sqrt_ratio` is currently used while computing `hash_to_curve`.

Recommendation Add a constraint that `num` should not be zero

4.1.9 V-CSB-VUL-009: hash_to_field msg could be unconstrained

Severity	Low	**Commit**	0350dd7
Type	Under-constrained Cell	**Status**	Fixed
File(s)	`ecc/hash_to_curve.rs`		
Location(s)	expand_message()		
Confirmed Fix At	196f66a		

Axiom's Halo2 library allows users to specify how a new cell should be constrained when added to the plonk table with a `QuantumCell`. In particular, a `Constant` cell will constrain the value of the cell to be equal to the input constant, an `Existing` cell will constrain the value of the cell to be equal to another cell and finally a `Witness` cell will not be constrained. As such, `Witness` cells typically need further interaction to either constrain their values or make them public. In the case of `expand_message`, shown below, a user may pass in witness cells that have no external constraints.

```rust
fn expand_message<F: BigPrimeField, HC: HashInstructions<F>>(
    thread_pool: &mut HC::CircuitBuilder,
    hash_chip: &HC,
    range: &impl RangeInstructions<F>,
    msg: impl Iterator<Item = QuantumCell<F>>,
    dst: &[u8],
    len_in_bytes: usize,
) -> Result<Vec<AssignedValue<F>>, Error> {
    ...

    let assigned_msg = msg
        .map(|cell| match cell {
            QuantumCell::Existing(v) => v,
            QuantumCell::Witness(v) => thread_pool.main().load_witness(v),
            QuantumCell::Constant(v) => thread_pool.main().load_constant(v),
            _ => unreachable!(),
        })
        .collect_vec();

    ...
}
```

Snippet 4.7: Snippet from `expand_message`

Impact Should a user provide a list of witnesses, the input will be unconstrained and so the output will be unconstrained.

Recommendation Don't allow witness cells to be provided to `expand_message`.

4.1.10 V-CSB-VUL-010: Unchecked hash length assumption

Severity	Low	**Commit**	0350dd7
Type	Data Validation	**Status**	Fixed
File(s)	`ecc/hash_to_curve.rs`		
Location(s)	expand_message()		
Confirmed Fix At	bddd0f4		

Per the reference on which the `expand_message` function is based, the return value should contain at least 32 bytes worth of data to be secure. No such check is done in this function — at most `len_in_bytes` bytes are taken from the result of the hash digest. No check is ever performed to ensure the hash digest has at least `len_in_bytes` bytes.

Impact This may lead to unexpected behavior or weaker guarantees.

Recommendation Add a check that the hash digest length contains at least the required number of bytes.

4.1.11 V-CSB-VUL-011: Missing check for expand_message

Severity	Low	**Commit**	0350dd7
Type	Data Validation	**Status**	Fixed
File(s)	`ecc/hash_to_curve.rs`		
Location(s)	expand_message()		
Confirmed Fix At	bddd0f4		

According to the reference implementation, the function `expand_message` should abort if either `ell > 255` or `len_in_bytes > 65535` or `len(DST) > 255`. No such checks occur in the current implementation.

Impact Unexpected behavior may occur if the function is used when these assumptions are not satisfied.

Recommendation Add checks that these assumptions are satisfied.

4.1.12 V-CSB-VUL-012: Missing equality check

Severity	Low	**Commit**	0350dd7
Type	Data Validation	**Status**	Fixed
File(s)	`ecc/hash_to_curve.rs`		
Location(s)	map_to_curve()		
Confirmed Fix At	f41e0bd		

The `add_unequal` function adds two elliptic curve points, but it does not support adding two points that share the same x-coordinate. It will explicitly check for this condition if the last input to the function is `true` but otherwise that check will be excluded. The `map_to_curve` function, shown below, omits this check. While it's unlikely that the inputs to `add_unequal`, namely `p1` and `p2`, are the same it is still possible.

```rust
fn map_to_curve(
    &self,
    ctx: &mut Context<F>,
    u: [FC::FieldPoint; 2],
) -> Result<EcPoint<F, FC::FieldPoint>, Error> {
    let [u0, u1] = u;

    let p1 = self.map_to_curve_simple_swu(ctx, u0);
    let p2 = self.map_to_curve_simple_swu(ctx, u1);

    let p_sum = self.ecc_chip.add_unequal(ctx, p1, p2, false);

    let iso_p = self.ecc_chip.isogeny_map(ctx, p_sum);

    Ok(self.ecc_chip.clear_cofactor(ctx, iso_p))
}
```

Snippet 4.8: Snippet from `map_to_curve`

Impact In the case where `p1 = p2`, the result of `add_unequal` will be under-constrained and could be manipulated by an attacker.

Recommendation Make the last argument to `add_unequal` `true` in this case.

4.1.13 V-CSB-VUL-013: Consider adding must_use macro

Severity	Warning	**Commit**	0350dd7
Type	Maintainability	**Status**	Fixed
File(s)	Multiple		
Location(s)	N/A		
Confirmed Fix At	66bd80d		

Several functions, including the `is_valid_signature` function shown below, return values that must be constrained by the user. If the return value is ignored for these functions, the added constraints will effectively be useless as they only serve to compute the returned value. It is therefore likely that in this case the user has incorrectly used the API and it should either be removed or a logic error exists.

```rust
pub fn is_valid_signature(
    &self,
    ctx: &mut Context<F>,
    signature: EcPoint<F, FieldVector<ProperCrtUint<F>>>,
    msghash: EcPoint<F, FieldVector<ProperCrtUint<F>>>,
    pubkey: EcPoint<F, ProperCrtUint<F>>,
) -> AssignedValue<F> {
    let g1_chip = EccChip::new(self.fp_chip);

    let g1_neg = g1_chip.assign_constant_point(ctx, G1Affine::generator().neg());

    let gt = self.compute_pairing(ctx, signature, msghash, pubkey, g1_neg);

    let fp12_chip = Fp12Chip::<F>::new(self.fp_chip);
    let fp12_one = fp12_chip.load_constant(ctx, Fq12::one());

    fp12_chip.is_equal(ctx, gt, fp12_one)
}
```

Snippet 4.9: Definition of the `is_valid_signature` function

Recommendation For these functions, add the `#[must_use]` functions to prevent potential errors. This macro will emit a warning at compilation time if the return value is not used.

4.1.14 V-CSB-VUL-014: User must determine sign of ratio root

Severity	Warning	**Commit**	0350dd7
Type	Under-constrained Cell	**Status**	Fixed
File(s)	`ecc/hash_to_curve.rs`		
Location(s)	sqrt_ratio()		
Confirmed Fix At	8769766		

When determining the square root of the input ratio, `sqrt_ratio` first computes the value of the resulting square as shown below. It then compares the square of the return value (`y_sqr`) to the actual value of the square (`y_check`). There are, however, two values that could result in the same `y_sqr`, namely `y_assigned` and its negation. The return value `y_assigned` is therefore under-constrained since it is not constrained to have the appropriate signed.

```
fn sqrt_ratio(
    &self,
    ctx: &mut Context<F>,
    num: FC::FieldPoint,
    div: FC::FieldPoint,
) -> (AssignedValue<F>, FC::FieldPoint) {
    ...

    let y_assigned = field_chip.load_private(ctx, y);
    let y_sqr = field_chip.mul(ctx, y_assigned.clone(), y_assigned.clone()); // y_sqr
     = y1^2

    ...

    field_chip.assert_equal(ctx, y_check, y_sqr); // assert y_check == y_sqr

    (is_square, y_assigned)
}
```

Snippet 4.10: Definition of the `sqrt_ratio` function

Impact Should the caller not constrain the sign of `y_assigned`, it may be unconstrained. It should be noted that all of the code that uses `sqrt_ratio` does resolve the sign of `y` but documentation should be added so future users of the function do so as well.

Recommendation Document the assumption that the caller must constrain the sign of `y`.

4.1.15 V-CSB-VUL-015: Misleading, outdated, missing or wrong comments

Severity	Warning	**Commit**	0350dd7
Type	Maintainability	**Status**	Fixed
File(s)	Multiple		
Location(s)	N/A		
Confirmed Fix At	da31478		

In several places the code seems to have been revised after comments have been added. Hence the current comments are wrong or not up-to-date. Some comments have been directly incorporated verbatim from other places, but do not apply. In some places additional assumptions have not been state precisely in the comments. Some variable and function names can be misleading.

Here a list of places which could benefit from careful revision:

- `bls12_381/bls_signature.rs`
  - It is indicated that verification is done by checking `e(g1, signature)*e(pubkey, -H(m)) === 1`, whereas it checks `e(-g1, signature)*e(pubkey, H(m)) === 1` (which is of course equivalent).
- `bls12_381/pairing.rs`
  - The comment for function `sparse_line_function_unequal` does not reflect the appropriate variable names and output order.
  - Line 42: `out4` is the coefficient of w^2 and would better be called `out2`.
  - Line 43: `out2` is the coefficient of w^5 and would better be called `out5`.
  - line 80: `out2` is the coefficient of w^4 and would better be called `out4`.
  - The comments for function `sparse_line_function_equal` state that `Q = (x, y) is a point in E(Fp) should be in E(Fp2)` and `P = (P.x, P.y) in E(Fp2) should be in E(Fp)`. The comments then give the expression `(3x^3 - 2y^2)(XI_0 + u) + w^4 (-3 x^2 * Q.x) + w^3 (2 y * Q.y)` as part of the formula for computing the output values. In this equation `x` and `y` should be denoted `Q.x` and `Q.y` and `Q.x` and `Q.y` should be `P.x` and `P.y`.
  - The comments for function `fp12_multiply_with_line_unequal` confuse `P` and `Q`. In particular, `P`, `P0`, and `P1` should be `Q`, `Q0`, and `Q1`. Furthermore, there is an unstated assumption that `Q0 != Q1` which should be added to the comments.
- `bls12_381/hash_to_curve.rs`
  - Line 156: `Plet` should be `P`.
  - The function name `mul_by_bls_x` is confusing — this function multiplies by `-x` , i.e., by `0xd201000000010000` . Either the name should be changed or details specified as a comment.

Impact Inadequate comments hinder the understanding of the expected behavior and impede the re-usability and maintenance of the code. Misunderstanding in the behavior might yield bugs when code is revised / reused. Use of functions while unaware of implicit assumptions can cause further vulnerabilities.

Recommendation Revise unclear code comments to improve understanding of the code and prevent code misuse in the future.

4.1.16 V-CSB-VUL-016: Undocumented non-zero input assumption

Severity	Warning	**Commit**	0350dd7
Type	Maintainability	**Status**	Fixed
File(s)	`bls12_381/final_exp.rs`		
Location(s)	cyclotomic_decompress(), final_exp()		
Confirmed Fix At	504f771		

The `divide_unsafe` function allows a user to divide two values but is undefined in cases where the denominator is `0`. Rather than explicitly checking that this property holds, the caller must ensure that the denominator cannot be `0`. Two functions, `cyclomatic_decompress` and `final_exp`, shown below, assume that some inputs are non-zero but do not explicitly state that assumption.

```
pub fn cyclotomic_decompress(
    &self,
    ctx: &mut Context<F>,
    compression: Vec<FqPoint<F>>,
) -> FqPoint<F> {
    let [g2, g3, g4, g5]: [_; 4] = compression.try_into().unwrap();

    ...

    let g1_0 = fp2_chip.divide_unsafe(ctx, &g1_num, &g3);
    let g2_is_zero = fp2_chip.is_zero(ctx, &g2);
    // resulting ‘g1‘ is already in "carried" format (witness is in ‘[0, p)‘)
    let g1 = fp2_chip.0.select(ctx, g1_0, g1_1, g2_is_zero);

    ...
}
```

Snippet 4.11: Snippet from the `cyclomatic_decompress` function

In the `cyclotomic_decompress` function, shown above, the function assumes that `g2` and `g3` cannot simultaneously be `0`. While this will hold if the inputs are valid (i.e. computed by `cyclomatic_compress`), in cases where the input comes from an unknown source this must be validated.

```
pub fn final_exp(
    &self,
    ctx: &mut Context<F>,
    a: <Self as FieldChip<F>>::FieldPoint,
) -> <Self as FieldChip<F>>::FieldPoint {
    // a^{q^6} = conjugate of a
    let f1 = self.conjugate(ctx, a.clone());
    let f2 = self.divide_unsafe(ctx, &f1, a);

    ...
}
```

Snippet 4.12: Snippet from the `final_exp` function

Similarly, the `final_exp` function shown above assumes that the input `a` is not equal to `0`. Currently, the function's input is computed by `miller_loop` which should return a non-zero value. If this function input should ever come from an unknown source or a source that does not have the same properties, however, it is possible for the output to be under-constrained.

Impact In both of these cases, it is possible for the output of the functions to be under-constrained.

Recommendation We would recommend either explicitly documenting the assumptions so that future users of the API perform the validation as necessary, or consider add the necessary zero-checks to the function.

4.1.17 V-CSB-VUL-017: Unnecessary clones

Severity	Info	Commit	0350dd7
Type	Maintainability	Status	Fixed
File(s)	`bls12_381/pairing.rs, ecc/hash_to_curve.rs`		
Location(s)	batched_pairing(), psi()		
Confirmed Fix At	2b793af		

The following line in `batched_pairing` unnecessarily clones the `mml` value (line 432):

```
let fe = fp12_chip.final_exp(ctx, mml.clone());
```

Snippet 4.13: Snippet from `batched_pairing`

Additionally, the following lines unnecessarily clone `psi_x` and `psi_y` in the function `psi` (lines 165-166):

```
let x = self.field_chip().mul(ctx, x_frob, psi_x.clone());
let y = self.field_chip().mul(ctx, y_frob, psi_y.clone());
```

Snippet 4.14: Snippet from `psi`

Finally, `psi2_x` is unnecessarily cloned in `psi2` (line 180) as shown in the snippet below:

```
let x = self.field_chip().mul(ctx, p.x, psi2_x.clone());
```

Snippet 4.15: Snippet from `psi2`

Impact This logic is inefficient and may confuse future developers.

Recommendation Remove unnecessary clones.