Symbiotic BLS12-381 Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Farouk

qpzm

December 11, 2025

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **2**

6 Executive Summary **3**

7 Findings **5**

7.1 Informational 5

7.1.1 `BLS12381::findYFromX` returns `sqrt(-a)` instead of reverting when input is not a quadratic residue 5

7.1.2 Inconsistent handling of point at infinity between `isOnCurve` and `isInSubgroup` 6

7.2 Gas Optimization 7

7.2.1 `BLS12381::hashToG1` can use constants for DST 7

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The BLS12381 library implements the full set of arithmetic and hashing primitives required to operate over the BLS12-381 G1 subgroup, exposing a deterministic and self-contained interface for point construction, mapping, and validation. The contract defines the curve using its affine Weierstrass form over the base field Fp, encoding G1 elements as two-limb field values for both x and y coordinates. The point at infinity is explicitly represented as the all-zero tuple, allowing group operations and precompile interactions to treat it as the identity element.

Hashing a message to the curve follows the structure of the hash-to-curve standard. The message is expanded using SHA-256 in XMD mode, producing a sequence of pseudo-random bytes that are reduced into valid Fp field elements. These elements are then mapped to the curve using the Ethereum precompile for field-to-curve mapping, which performs the Simplified SWU map followed by an isogeny and full cofactor clearing to ensure the resulting point lies in the correct subgroup. The mapped points are added to yield the final hash-to-G1 output.

The library also includes functionality for reconstructing points from x-coordinates using the standard exponentiation-based square-root method valid for primes where p mod 4 = 3. Field operations are expressed in a limb-wise format compatible with EVM word-level arithmetic, including carry handling, modular adjustments, and controlled use of preallocated memory. Group-theoretic operations such as negation, scalar multiplication, and subgroup membership checks are structured around these same primitives and align with the curve's algebraic definition.

Together, these components provide the underlying mathematical operations needed by higher-level key and signature modules, offering deterministic hashing, subgroup-sound point generation, and consistent interaction with Ethereum's pairing and mapping precompiles.

5 Audit Scope

Cyfrin conducted an audit of the BLS12-381 contract for Symbiotic based on the code present in the repository commit hash 9b9397f. The following file construct the scope of the audit:

- src/libraries/utils/BLS12381.sol

6 Executive Summary

Over the course of 3 days, the Cyfrin team conducted an audit on the Symbiotic BLS12-381 smart contracts provided by Symbiotic. In this period, a total of 3 issues were found.

This audit assessed the standalone BLS12381 library, which implements the low-level arithmetic, hashing, and point-validation routines required for BLS12-381 signatures. The review covered field operations, square-root extraction, hash-to-curve logic, memory-safe assembly usage, and the encoding of the point at infinity. The codebase is generally well-structured, mathematically consistent, and its memory-level routines are safely implemented, with all assembly blocks either initializing fresh memory via `mload(0x40)` or writing only inside pre-allocated buffers.

Several non-critical issues were identified. The hashing layer currently reconstructs the domain-separation tag on every `hashToG1` call, leading to unnecessary memory allocation; embedding the DST as constants avoids this overhead. The `findYFromX` routine correctly implements the square-root formula for primes where `p mod 4 = 3`, but, as expected from the algebra, returns `sqrt(-a)` when the input is not a quadratic residue. While upstream validation prevents malformed inputs from propagating, explicitly handling this case would avoid silent construction of invalid points in other contexts. The point-validation helpers also treat the point at infinity inconsistently: the identity element is accepted by subgroup checks and handled correctly by operations like addition and negation, but is rejected by `isOnCurve` despite being a valid group element under this encoding. Aligning these checks improves clarity for integrators.

Beyond these observations, the mathematical routines operate as intended. Negation handles both borrow and no-borrow cases correctly, `_xCubePlus4` manages carries and modular reduction safely, and the `expandMsg` implementation matches the RFC 9380 structure for XMD-SHA-256. No security-relevant issues were found in this library. The recommended adjustments primarily strengthen consistency, correctness on edge cases, and gas efficiency, without affecting the library's safety as currently integrated.

Summary

Project Name	Symbiotic BLS12-381
Repository	relay-contracts
Commit	9b9397f97296...
Audit Timeline	Dec 1st - Dec 3rd
Methods	Manual Review, Differential Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	0
Informational	2
Gas Optimizations	1
Total Issues	3

Summary of Findings

[I-1] `BLS12381::findYFromX` returns `sqrt(-a)` instead of reverting when input is not a quadratic residue	Acknowledged
[I-2] Inconsistent handling of point at infinity between `isOnCurve` and `isInSubgroup`	Resolved
[G-1] `BLS12381::hashToG1` can use constants for DST	Acknowledged

7 Findings

7.1 Informational

7.1.1 `BLS12381::findYFromX` returns `sqrt(-a)` instead of reverting when input is not a quadratic residue

Description: `BLS12381::findYFromX` computes the square root using the formula `y = (x^3+4)^((p+1)/4) mod p`. This formula only works correctly when `x^3+4` is a quadratic residue modulo p. When `x^3+4` is NOT a quadratic residue, the function returns `sqrt(-(x^3+4))` instead of reverting.

```
function findYFromX(uint256 x_a, uint256 x_b) internal view returns (uint256 y_a, uint256 y_b) {
    // compute (x**3 + 4) mod p
    (y_a, y_b) = _xCubePlus4(x_a, x_b);

    // compute y = sqrt(x**3 + 4) mod p = (x**3 + 4)^(p+1)/4 mod p
    // ...
}
```

By Euler's criterion:

- If `a` is a quadratic residue: `a^((p-1)/2) = 1 (mod p)`
- If `a` is NOT a quadratic residue: `a^((p-1)/2) = -1 (mod p)`

So when `a` is not a quadratic residue:

```
(a^((p+1)/4))^2 = a^((p+1)/2) = a^((p-1)/2) * a = (-1) * a = -a (mod p)
```

The result `a^((p+1)/4)` gives `sqrt(-a)`, not `sqrt(a)`.

Impact: `findYFromX` is used in `KeyBlsBls12381::deserialize`, If an invalid x-coordinate is provided where `x^3+4` is not a quadratic residue, `deserialize` returns an invalid point that is NOT on the curve, rather than reverting.

The impact is limited in the current codebase because:

1. In `KeyRegistry::_setKey`, keys are validated via `fromBytes` -> `wrap` which checks `isOnCurve` and `isInSubgroup` before storing
2. `deserialize` only reads from trusted storage containing validated keys
3. BLS precompiles: BLS12_G1ADD, BLS12_G1MSM, BLS12_PAIRING_CHECK would reject invalid points. https://github.com/ethereum/go-ethereum/blob/6f2cbb7a27ba7e62b0bdb2090755ef0d271714be/core/vm/contracts.go#L1211

However, if `findYFromX` or `deserialize` is used in another context without proper validation, it could return invalid curve points silently.

Proof of Concept: Example with p = 19 (where 19 = 3 (mod 4)):

a	Is QR?	a^5 mod 19	(a^5)^2 mod 19	Expected
4	Yes	17	4	a
3	No	15	16	-a = -3 = 16
2	No	13	17	-a = -2 = 17
5	Yes	9	5	a

For non-quadratic residues, `a^((p+1)/4)` gives `sqrt(-a)` instead of `sqrt(a)`.

Recommended Mitigation: For future upgrades that do not enforce curve membership and subgroup validation, ensure proper handling of the edge case where x^3+4 fails to be a quadratic residue.

Symbiotic: Acknowledged, as the team aims to leave `BLS12381` behavior similar to `BN254`.

7.1.2 Inconsistent handling of point at infinity between `isOnCurve` and `isInSubgroup`

Description: The `BLS12381::isOnCurve` function returns `false` for the point at infinity `(0, 0, 0, 0)`, while `isInSubgroup` returns `true` for the same point. This creates an inconsistency in how the identity element is validated.

In `isOnCurve`, the function checks if $y^2 \equiv x^3 + 4 \pmod{p}$:

- For `(0, 0, 0, 0)`: $y^2 = 0$ but $x^3 + 4 = 4$
- Since $0 \neq 4$, the function returns `false`

However, in `isInSubgroup`:

```
function isInSubgroup(G1Point memory point) internal view returns (bool) {
    G1Point memory result = scalar_mul(point, G1_SUBGROUP_ORDER);
    return result.x_a == 0 && result.x_b == 0 && result.y_a == 0 && result.y_b == 0;
}
```

The point at infinity correctly returns `true` since `0 * order = 0` (identity multiplied by any scalar is identity).

Mathematically, the point at infinity is a valid group element (the identity) but does not have affine coordinates satisfying the Weierstrass curve equation.

Note that other functions in the library handle the point at infinity correctly:

- `add`, `scalar_mul`, `pairing`: Handled by EIP-2537 precompiles
- `negate`: Explicit check returns `(0, 0, 0, 0)` unchanged
- `isInSubgroup`: Returns `true` correctly

This makes `isOnCurve` the sole outlier in its handling of the identity element.

Impact:

- If `isOnCurve` is used to validate G1 points before cryptographic operations, the identity element would be incorrectly rejected

Proof of Concept:

```
G1Point memory infinity = G1Point(0, 0, 0, 0);

bool onCurve = BLS12381.isOnCurve(infinity);      // returns false
bool inSubgroup = BLS12381.isInSubgroup(infinity); // returns true

// Inconsistent: valid subgroup element fails curve check
```

Recommended Mitigation: Add a special case check for the point at infinity in `isOnCurve`:

```
function isOnCurve(G1Point memory point) internal view returns (bool) {
    // Point at infinity is a valid curve point (identity element)
    if (point.x_a == 0 && point.x_b == 0 && point.y_a == 0 && point.y_b == 0) {
        return true;
    }
    // ... rest of the function
}
```

Symbiotic: Fixed in bedb2ea.

Cyfrin: Verified.

7.2 Gas Optimization

7.2.1 `BLS12381::hashToG1` can use constants for DST

Description: In `BLS12381::hashToG1`, the Domain Separation Tag (DST) string is passed as an inline string literal to `expandMsg`, causing unnecessary memory allocation on every call.

```
function hashToG1(bytes memory message) internal view returns (G1Point memory result) {
    bytes memory uniform_bytes = expandMsg("BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_NUL_", message,
    ↪  0x80);
```

https://github.com/symbioticfi/relay-contracts/blob/main/src/libraries/utils/BLS12381.sol#L287

At runtime, this:

1. Allocates 43 bytes in memory for the string
2. Copies the string literal from bytecode to memory
3. Incurs memory expansion costs
4. Passes a memory pointer to `expandMsg`

Impact: Gas is wasted on every `hashToG1` call. Since BLS signature verification is a common operation, this adds up.

Test	Before	After	Savings
`test_VerifyValidSignature`	401,631	400,494	1,137 gas
`test_HashToG1_OnCurveAndNonZero`	20,672	20,298	374 gas

Recommended Mitigation: Define the DST as constants and create a specialized `expandMsgBLS` function:

```
bytes32 private constant DST_PART1 = "BLS_SIG_BLS12381G1_XMD:SHA-256_S";
bytes11 private constant DST_PART2 = "SWU_RO_NUL_";
uint8 private constant DST_LEN = 43;

function hashToG1(bytes memory message) internal view returns (G1Point memory result) {
    bytes memory uniform_bytes = expandMsgBLS(message, 0x80);
    // ...
}

function expandMsgBLS(bytes memory message, uint8 n_bytes) internal pure returns (bytes memory) {
    bytes memory zpad = new bytes(0x40);
    bytes memory b_0 = abi.encodePacked(zpad, message, uint8(0x00), n_bytes, uint8(0x00), DST_PART1,
    ↪  DST_PART2, DST_LEN);
    // ... rest of expandMsg logic using DST_PART1, DST_PART2, DST_LEN
}
```

This embeds the DST directly in bytecode, avoiding runtime memory allocation.

Symbiotic: Acknowledged, as the team prefers to not perform changes on the `expandMsg`.