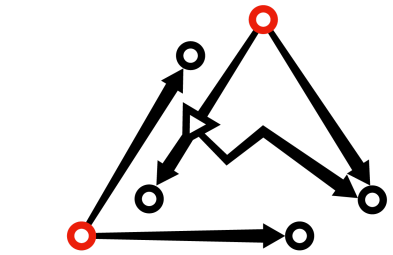

Least Authority

PRIVACY MATTERS

Sylow

Security Audit Report

Warlock

Final Audit Report: 6 January 2025

Table of Contents

Overview

Background

Warlock has requested that Least Authority perform a security audit of `Sylow` and `SolBLS`. `Sylow` is a Rust library for elliptic curve cryptography, specifically tailored for the `BN254` curve, and `SolBLS` is a Solidity library optimized for on-chain BLS signature verification.

Project Dates

- **October 7, 2024 - October 22, 2024**: Initial Code Review *(Completed)*
- **October 24, 2024**: Delivery of Initial Audit Report *(Completed)*
- **October 25, 2024**: Delivery of Updated Initial Audit Report *(Completed)*
- **January 3, 2024:** Verification Review *(Completed)*
- **January 6, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Will Sklenars, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the `Sylow` and `SolBLS` followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- `Sylow`: https://github.com/warlock-labs/sylow
- `SolBLS`: https://github.com/warlock-labs/solbls

Specifically, we examined the following Git revisions for our initial review:

- `Sylow`: `0f9f55adc3657299edcee9db031896c11b0b2782`
- `SolBLS`: `1ac7407a38df580d3a678aa4b9701667c387ef04`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- `Sylow`: https://github.com/LeastAuthority/warlock-labs-sylow
- `SolBLS`: https://github.com/LeastAuthority/warlock-labs-solbls

For the verification, we examined the following Git revisions:

- `Sylow`: `ae0cc6fb85bb20d5f3e4b5a92e1d3edfe7b7c0e1`
- `SolBLS`: `a7894c69defabb076a8de0154f4abbd044c3a94f`

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Dev Guide: https://x.com/warlock_xyz/status/1833173619985555676?s=46
- Website: https://warlock.xyz
- SolBLS - Zellic Audit Report Draft.pdf *(shared with Least Authority via Telegram on 21 October 2024)*

In addition, this audit report references the following documents:

- D. F. Aranha, P. S. L. M. Barreto, P. Longa, and J. E. Ricardini, “The Realm of Pairings.” *IACR Cryptology ePrint Archive*, 2013, [ABL+13]
- P. S. L. M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order." *IACR Cryptology ePrint Archive*, 2005, [BN05]
- J. Beuchat, J. E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya, “High-Speed Software Implementation of the Optimal Ate Pairing over Barreto–Naehrig Curves.” *IACR Cryptology ePrint Archive*, 2010, [BGM+10]
- D. Brumley and D. Boneh, “Remote Timing Attacks are Practical.”*USENIX Security Symposium*, 2003, [BB03]
- J. Chávez-Saab, F. Rodríguez-Henríquez, and M. Tibouchi, “SwiftEC: Shallue–van de Woestijne Indifferentiable Function To Elliptic Curves.” *IACR Cryptology ePrint Archive*, 2022, [CRT22]
- M. Joye, “Elliptic Curves and Side-Channel Analysis.” *ST Journal of System Research*, 2003, [Joye03]
- P. Longa, "Efficient Algorithms for Large Prime Characteristic Fields and Their Application to Bilinear Pairings." *IACR Cryptology ePrint Archive*, 2022, [Longa22]
- A. Menezes, P. V. Oorschot, and S. Vanstone, “Chapter 14: Efficient Implementation.” In *Handbook of Applied Cryptography*. *CRC Press*, 1996, [MOV96]
- H. Prodinger, "On Binary Representations of Integers With Digits -1, 0, 1." *Colgate University*, 2000, [Prodinger00]
- J. Renes, C. Costello, and L. Batina, "Complete addition formulas for prime order elliptic curves." *IACR Cryptology ePrint Archive*, 2015, [RCB15]
- A. Shallue and C. E. V. D. Woestijne, "Construction of rational points on elliptic curves over finite fields." *ACM Digital Library*, 2006, [SW06]
- Pedersen Hash – iden3 0.1 documentation: https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/pedersen-hash/pedersen.html
- Sylow: Elliptic Curve Cryptography Suite for BN254: https://docs.rs/sylow/latest/sylow
- EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128: https://eips.ethereum.org/EIPS/eip-197
- RFC 9380 | Hashing to Elliptic Curves: https://www.rfc-editor.org/rfc/rfc9380.html
- OWASP Risk Rating Methodology: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;

- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase

Findings

General Comments

Our team performed a security audit of `Sylow`, a Rust-based cryptographic library that implements cryptographic methods for the BLS signature scheme over the BN254 curve and `SolBLS`, which is a smart contract for BLS signature over the same curve.

System Design

Our team reviewed `Sylow` and `SolBLS` and found the system to be well-designed, with a strong emphasis on security. We audited the code under the assumption that it can potentially be used by third parties different from the Warlock team.

Sylow

The `Sylow` repository is overall well-designed with two crates for handling field and group operations respectively. It also contains dedicated modules for performing hash operations, point mapping operations using the Shallue-van de Woestijne method [SW06], and elliptic curve pairing operations, among others.

We found that the code generally follows security best practices. However, during our review we identified a few issues, which we summarize below:

We found that the `is_square` function in `fields/fp2.rs` is implemented incorrectly (Issue A) and that this was not detected by the tests currently implemented in the codebase (Issue D). We additionally found that the use of random point generation in `groups/g2.rs` potentially exposes discrete log relations among random elements in $\mathbb{G}_2$ (Issue B). We also found that the scalar multiplication algorithm implemented in `groups/group.rs` follows a double-and-add algorithm that is not resistant against timing attacks, which can potentially leak the secret key (Issue E). Our team further noted that secret keys might be recoverable from unprotected memory (Issue C).

Furthermore, we noticed that the function `inverse` computes the inverse of a base field element `x` using Fermat's little theorem (i.e., `x^(-1) = x^(p-2) for x != 0`), but has no check to throw an error in case `x = 0`. This is expected behavior, as it is required by RFC 9380. Our team did not identify any issues relating to this area of investigation.

In `fields/fp2.rs`, we found that certain functions are implemented specifically with the quadratic non-residue set to β=-1, which is suitable for the case of the curve BN254; however, this value will not be valid for other choices of β or curves. We recommend using the quadratic non-residue as a parameter to support generic choices of β or curves (Suggestion 2).

Additionally, we compared the code against the specified algorithms and found two bugs in [BGM+10], Algorithm 17 (Suggestion 6).

SolBLS

During our investigation of `SolBLS`, we reviewed the smart contracts for potential exposure to re-entrancy attacks, and did not identify any issues. While `BLS.sol` does contain some calls to external contracts, these contracts are Ethereum precompiles – specifically the precompiles at address `0x05` (`Modexp`), address `0x06` (`ecAdd`), and address `0x08` (`ecPairing`) – and do not pose significant exposure to re-entrancy attacks.

We observe that the naming of the function `isValidSignature` is misleading since it only checks whether the input point is in the group $\mathbb{G}_1$. We suggest updating the function either by renaming it or changing the functionality (Suggestion 7).

We noted that the function `verifySingle` does not perform input validation in order to check that the arguments `signature`, `pubkey`, and `message` are on the curves $\mathbb{G}_1$ and $\mathbb{G}_2$. These inputs are used to call the `ecPairing` precompile at address `0x08`. We checked the precompile specification and found that the precompile reverts if inputs are invalid. It is therefore not required for the `verifySingle` function to perform the validations. However, we found one area where input validations could be improved to better adhere to technical specifications (Issue G).

Code Quality

We performed a manual review of the repositories in scope and found the code to be clean, well-organized, and of high quality, in that it adheres closely to development best practices.

Tests

The repositories in scope include some tests; however, we found that test coverage, especially with regards to some of the complex functionalities, can be improved. Our team also noted that Issue A could have been detected if tests were implemented to check for it (Issue D). Additionally, although `SolBLS` has sufficient test coverage overall, it could be improved with more unit testing.

Documentation and Code Comments

The project documentation provided by the Warlock team is comprehensive and sufficiently describes the intended functionality of the system. However, we found some typographical errors and inconsistencies in the code documentation, which we recommend be corrected and updated (Suggestion 3).

Moreover, we found that code comments sufficiently describe the intended behavior of security-critical components and functions and serve as an additional, helpful source of documentation. Our team additionally identified further opportunities for improving the code comments (Suggestion 4 and Suggestion 6).

Scope

The scope of this review was sufficient and included all security-critical components.

Dependencies

We examined all the dependencies implemented in the codebase and identified two issues: RUSTSEC-2024-0375 (`atty` is unmaintained) and RUSTSEC-2021-0139 (`ansi_term` is unmaintained). The Warlock team is aware of these issues, as noted in Issue #39 and Issue #30.

Additionally, our team noted that `SolBLS` does not contain any dependencies.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Incorrect Square Check in `F_(p^2)`	Low	Resolved
Issue B: Exposed Discrete Log Relations Between Random Elements in $\mathbb{G}_2$	Low	Resolved
Issue C: Potential Exposure of Secret Keys	Medium	Resolved
Issue D: Insufficient Test Coverage	Low	Resolved
Issue E: Scalar Multiplication Vulnerable To Timing Attacks	Medium	Resolved
Issue F: Missing Subgroup Check	Low	Resolved
Issue G: Improve Validation for Domain Separation Tag length	Low	Resolved
Issue H: Incomplete Group Membership Check in G1Projective::new	High	Resolved
Suggestion 1: Implement Extension Field Defaults as Base Field Default Extension	Informational	Resolved
Suggestion 2: Make Codebase for Complex Extensions Generic Over Quadratic Non-Residue	Informational	Unresolved
Suggestion 3: Correct Typographical Errors and Incorrect References in Code Comments	Informational	Partially Resolved
Suggestion 4: Improve Code Comments	Informational	Resolved
Suggestion 5: Correct Tracing Variable	Informational	Resolved
Suggestion 6: Add Note Clarifying Bugs in Reference Article	Informational	Resolved
Suggestion 7: Rename Function To Match Functionality	Informational	Unresolved
Suggestion 8: Use Standard Terminology for Domain Separation Tag	Informational	Unresolved
Suggestion 9: Upgrade Solidity Version and Lock the Pragma	Informational	Unresolved

Issue A: Incorrect Square Check in `F_(p^2)`

Location

`src/fields/fp2.rs#L178`

Synopsis

Our team found an incorrect check for quadratic residues in the extension field `F_(p^2)` and identified a typographical error in its implementation.

Severity

Low.

Impact

The function is currently not called anywhere in the codebase.

Technical Details

The code logic should verify whether an element `b=(b0 + b1v) ∈ F_(p^2)` is a square; that is, whether another element `a=(a0 + a1v) ∈ F_(p^2)` exists with `a^2 = b`. In order to verify this, the Warlock team implemented a check to confirm that the expression `b0^2 + βb1^2` is a square in the base field, assuming that `(b0 + b1v)` is a square in `F_(p^2)` if and only if `b0^2 + βb1^2` is a square in $\mathbb{F}$. In our review, we found that a typographical coding error was introduced and that the Warlock team actually implements `b0^2 + βb0^2`, which for `β=-1` is always zero, leading to a situation where the team's insufficient tests for squares always passes since the Legendre symbol of zero is always zero.

However, the actual issue is not the typographical error but rather that the assumption that `(b0 + b1v)` is a square in `F_(p^2)` if and only if `b0^2 + βb1^2` is a square in $\mathbb{F}$ is incorrect. To further clarify, consider the counterexample `b0=0` and `b1=1`. Then `b0 + b1v` will have the following square root `(a0 + a1v) ∈ F_(p^2)`:

```
a0 =
19848962826107723227364695958474411417416760630979784781770308649712843043 88
a1=
19848962826107723227364695958474411417416760630979784781770308649712843043 88
```

However, the term `b0^2 + βb1^2 = β` is not a square in $\mathbb{F}$ by assumption on β. (Otherwise, `F_(p^2) = 𝔽₍(X) / (X² - β)` would not be an extension field).

On the code level, to see that the typographical error is not the actual issue, assume the code was corrected as follows:

```
sum = self.0[0].square() + FP_QUADRATIC_NON_RESIDUE * (-self.0[1]).square()
```

Then the following test would fail:

```
#[test]
fn test_is_square() {
  let b = create_field_extension([0, 0, 0, 0], [1, 0, 0, 0]);
  let is_square = b.is_square();
  assert!(bool::from(is_square), "b = [0, 1] should be a square in Fp2");
}
```

Remediation

We recommend implementing a proper squaring check in `F_(p^2)`.

Status

The Warlock team has removed this function from the codebase.

Verification

Resolved.

Issue B: Exposed Discrete Log Relations Between Random Elements in $\mathbb{G}_2$

Location

`src/groups/g2.rs#L227`

Synopsis

In the referenced part of the code, a random group element in the pairing group $\mathbb{G}_2$ is computed as `rand_g2 = [rand_f] * g`, where `rand_f` is randomness from the base field, and `g` is a generator of $\mathbb{G}_2$. This construction exposes discrete log relations between different random elements that are generated in this manner.

Severity

Low.

Impact

The severity of the impact depends on how the randomness generation is used in the code. Best practices recommend constructing randomness in algebraic structures that have no known relations between different instantiations.
This is crucial, for example, in Pederson hashes, where the generators in the group should be constructed in such a way that no discrete log relations can be computed between them (see here for more details). In Warlock's case, the function is to be used only for elliptic curve point generation and not as a (cryptographic) random number generator or as a pseudorandom function.

Remediation

We recommend refraining from using a random point generation that exposes discrete log relations.

Status

The Warlock team has reported that the function `rand` will not be used as a (cryptographic) random number generator, and has added detailed explanations in the comments, clarifying the correct use of the function. Hence, for this particular function in question, our team did not identify any issues.

Verification

Resolved.

Issue C: Potential Exposure of Secret Keys

Location

`src/lib.rs#L105`

Synopsis

Since the Warlock team implements a constant time codebase, it might therefore be possible for an attacker to gain physical access to the machine/computer. In such a case, secrets stored in non-protected memory pose additional risks.

Severity

Medium.

Impact

The impact depends on the threat model. If physical access or compromised machines are considered a threat, an attacker could gain access to secret keys either by inspection of the non-protected memory or due to potential leaks into logs.

Preconditions

This exploit requires physical access or a compromised machine.

Remediation

The Warlock team is aware of this issue. Our team agrees with the development team's comment that Rust's `secrets` crate can be integrated to remediate this issue.

Status

The Warlock team has updated the code to use the `secrets` crate to store generated key pairs and signatures.

Verification

Resolved.

Issue D: Insufficient Test Coverage

Location

`warlock-labs-sylow`

Synopsis

There is insufficient test coverage implemented to test the correctness of the implementation and that the system behaves as expected. Tests help identify coding implementation errors, which could lead to security vulnerabilities.

Sufficient test coverage should include tests for success and failure cases (all possible branches), which helps identify potential edge cases, and protect against errors and bugs that may lead to vulnerabilities. A test suite that includes sufficient coverage of unit tests and integration tests adheres to development best practices. In addition, end-to-end testing is also recommended to assess if the implementation behaves as intended.

Below, we list a few opportunities for improving test coverage:

- The tests in F_(p^6) are insufficient. Only a small amount of hard coded elements are used to test basic properties. We recommend using random elements with fixed seeds for predictability. Additionally, the Warlock team did not test edge cases and whether `x^2 = x*x` or `x* x^(-1) = 1` for `x != 0`.
- The tests in F_(p^12) are insufficient. There are no tests using random elements and `sparce_mul`, in particular, is not tested at all. We recommend increasing testing for `sparce_mul`, as it might have several errors due to its complexity. We also recommend running the tests on multiple random elements.

Severity

Low.

Impact

Insufficient tests could lead to potential edge cases and errors being missed, which may lead to vulnerabilities. In addition, the lack of sufficient test coverage would hinder the ability to assess whether the implementation behaves as intended.

Remediation

We recommend that comprehensive unit test coverage be implemented in order to identify any implementation errors and to verify that the implementation behaves as expected.

Status

The Warlock team has added tests for missing edge cases, as well as a basic implementation check for the sparse multiplication of elements in `F_(p^12)`, with an additional square check test. Furthermore, for the `F_(p^6)` and `F_(p^12)` extension fields, the Warlock team has removed the instantiation of concrete elements for the tests, wherever possible, in favor of arbitrary random ones.

Verification

Resolved.

Issue E: Scalar Multiplication Vulnerable To Timing Attacks

Location

`src/groups/group.rs#L649-L667`

Synopsis

In a scalar multiplication, an elliptic curve point P is multiplied by a scalar chosen from the underlying field. The code snippet referenced above implements a scalar multiplication that is not resistant against timing attacks. In particular, the code first derives a non-adjacent form (NAF) representation of the scalar. This is followed by a naive multiplication, which includes an iterative doubling operation and a conditional addition or subtraction operation, depending on the NAF bits. The conditional operation makes the code vulnerable to timing attacks (see [Joye03] and [BB03]).

Severity

Medium.

Impact

A successful timing attack reveals information about the scalar used, which can potentially correspond to the secret key of a signature.

Remediation

We recommend implementing a timing attack-resistant scalar multiplication algorithm. For example, a classic remediation is the Montgomery ladder [Joye03]:

```
function montgomery_ladder(k, P):
  R0 = Point at infinity
  R1 = P
  for bit in k (from most significant to least significant):
    if bit == 0:
      R1 = R0 + R1
      R0 = 2 * R0
    else:
      R0 = R0 + R1
      R1 = 2 * R1
```

```
return R0
```

Another possible solution would be to randomize the NAF representation of the scalar, each time the algorithm is called. This way, the algorithm would execute a different branching pattern at each call, effectively protecting against timing analysis.

Status

The Warlock team has implemented the recommended Montgomery Ladder approach for the scalar multiplication of elements in $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ using Algorithm 2c of [Joye03]. In addition, the team has adjusted the cyclotomic exponentiation in `pairing.rs` to use this same constant-time approach.

Verification

Resolved.

Issue F: Missing Subgroup Check

Location

`src/BLS.sol#L132`

Synopsis

The function `isValidPublicKey` only checks that the given data satisfies the curve equation of the quadratic twist curve of BN254, but does not check that the point is actually in $\mathbb{G}_2$.

Severity

Low.

Impact

The name `isValidPublicKey` suggests that the function correctly checks whether the given data represents a BLS public key; that is, an element of the pairing group $\mathbb{G}_2$. However, since the function only checks that the data is an element in the twist curve, keys that are outside of $\mathbb{G}_2$ would pass the test. Additionally, since those keys would not be usable in the EVM precompile for pairings (because the subgroup test is done in the precompile independently), it might lead to other consequences depending on where this function is used.

Preconditions

The attacker would need to generate a point in `BN254_twist( F_(p^2))\`$\mathbb{G}_2$, which is trivial.

Feasibility

Straightforward.

Remediation

We recommend implementing the cofactor check `r* P == 0` for any claimed public key `P` and group order `r`.

Status

The Warlock team has implemented the cofactor check by calling the precompiled contract at address 8 (the “BN256 pairing” precompile) that enforces the inputs to be in the appropriate subgroup.

Verification

Resolved.

Issue G: Improve Validation for Domain Separation Tag length

Location

`src/BLS.sol#L246`

Synopsis

RFC 9380, which describes the hashing of elliptic curves, specifies that the Domain Separation Tag (DST) should have a length between 1 and 255 bytes. Our team noted that `expandMsgTo96` correctly enforces the upper bound for DST length; however, no lower bound is enforced. Using an empty string as the DST does not throw an exception. This violates the RFC 9380 specification.

Severity

Low.

Impact

An empty domain separator could potentially allow replay attacks in other contracts.

Remediation

We recommend reverting if the DST length is `0`, in accordance with RFC 9380.

Status

The Warlock team has added an explicit check for a non-zero DST length, and an error is thrown when the DST length is zero.

Verification

Resolved.

Issue H: Incomplete Group Membership Check in G1Projective::new

Location

`src/groups/g1.rs#L383`

Synopsis

The `G1Projective::new(v: [Fp; 3])` method incorrectly checks group membership for the point at infinity, potentially allowing points on the line at infinity to pass the test.

Severity

High.

Impact

Points from the projective plane that do not satisfy the curve equation can have unintended consequences in the BLS signature scheme.

Technical Detail

In the `G1Projective::new(v: [Fp; 3])` method, the code checks if a given projective point `[X:Y:Z]` is a point on the BN254 curve using:

```
let is_on_curve = lhs.ct_eq(&rhs) | Choice::from(v[2].is_zero() as u8);
```

Here, `v[2]` corresponds to `Z` and the `is_on_curve` test passes for `Z==0`. Hence, any element of the

form `[X:Y:0]` will pass the check, which is incorrect since only the point at infinity `[0:Y:0]` but not the line at infinity `[X:Y:0]` should pass this branch of the test.

Remediation

We recommend implementing a test that checks that both `v[0]` and `v[z]` are zero.

Status

The Warlock team has modified the `is_on_curve` check to be passed by the point at infinity but not the line at infinity, by checking if both the X and Z coordinates are zero.

Verification

Resolved.

Suggestions

Suggestion 1: Implement Extension Field Default as Base Field Default Extension

Location

src/fields/extensions.rs#L189

Synopsis

The Warlock team has implemented the default of an extension field as a vector of defaults of the base field, while the default of an extension field should be implemented as the default of the base field instead. In Warlock's case, both approaches lead to the same result because the default of the base is defined as zero. However, for other values, the Warlock team's approach would lead to unexpected results. Therefore, it would be conceptually better to write:

```
fn default() -> Self {
  let mut retval = [F::zero(); N];
  retval[0] = F::default();
  Self::new(&retval)
}
```

This way, the default of any extension field would always be the default of the base field.

Mitigation

We recommend implementing the base field default extension as the extension field default.

Status

The Warlock team has updated the default function as suggested.

Verification

Resolved.

Suggestion 2: Make Codebase for Complex Extensions Generic Over Quadratic Non-Residue

Location

src/fields/fp2.rs

Synopsis

As explained in the description of the quadratic extension fields implementation, the codebase should be generic over a quadratic non-residue β, such that the associated extension field is defined as:

$$F_{p^2} = \mathbb{F}_p(X) / (X^2 - \beta)$$

However, in some instances in the codebase, it is assumed that $\beta = -1$; therefore, the code would not be valid for other values of β. While this approach may be appropriate in the case of BN254 because -1 is a quadratic non-residue in the associated base field, it might not work for other curves or choices of β.

Mitigation

We recommend making the code fully generic over the quadratic non-residue β.

Status

The Warlock team has stated that this change is harder to implement in practice. The team added that they will eventually support multiple curves, but doing so will require both algorithmic and structural changes to the code that are out of scope for this engagement.

Verification

Unresolved.

Suggestion 3: Correct Typographical Errors and Incorrect References in Code Comments

Location

Examples (non-exhaustive):

src/groups/group.rs#L340-L342

src/groups/group.rs#L536

src/groups/group.rs#L98

src/svdw.rs

src/pairing.rs#L496-L497

src/fields/fp6.rs#L283

src/pairing.rs#L756

src/pairing.rs#L798

src/fields/fp6.rs#L3

src/groups/g2.rs#L121

Synopsis

During our review, our team identified typographical errors and incorrect references in the code comments that impact the quality, readability, and maintainability of the codebase. To illustrate, the following is a non-exhaustive list of examples:

- In `src/groups/group.rs#L340-L342`, the reference corresponding to Algorithm 9 for doubling is `[2]`, not `[1]`.
- In `src/groups/group.rs#L536`, similarly, the reference corresponding to Algorithm `10` for addition is `[2]`, not `[1]`.
- In `src/groups/group.rs#L98`, the equation should be updated to `(x', y') |-> (w^2x', w^3y')`.
- In `src/svdw.rs`, although the main concepts are explained in the provided reference present in the comment, the algorithmic steps followed in `find_z_svdw`, `precompute_constants` and `unchecked_map_to_point` were referred to from RFC 9380, Section 6.6.1. We recommend adding this reference to the relevant locations.
- In `src/pairing.rs#L496-L497`, there are some typographical errors that should be corrected.
- In `src/fields/fp6.rs#L283`, the referenced algorithm is incorrect. It should be equation 9, page 14 from [Longa22].
- In `src/pairing.rs#L756`, the `doubling_step` follows equation 11 of [ABL+13]. The reference in the comment is currently incorrect.
- In `src/pairing.rs#L798`, the `addition_step` follows equation 12 of [ABL+13]. The reference in the comment is currently incorrect.

Mitigation

We recommend correcting the errors in the code comments by addressing the items listed above.

Status

The Warlock team has corrected most of the typographical errors. However, some errors still remain in the following locations:

- In `src/pairing.rs#L503`, “betwixt” should be updated to “between.”
- In `src/fields/fp6.rs#L297`, “Algo 9” should be updated to “Equation 9.”

Verification

Partially Resolved.

Suggestion 4: Improve Code Comments

Location

Examples (non-exhaustive):

`src/fields/fp.rs#L653`

`src/groups/g1.rs#L165-168`

Synopsis

While most of the code comments are comprehensive and helpful, we did identify a few opportunities for improvement. Below, we list some of our findings:

- In `src/fields/fp.rs#L653`, the `compute_naf` function uses the Prodinger algorithm correctly. However, it is not clear where the algorithm is explained in [Prodinger00]. A more exact reference of the Prodinger algorithm should be provided.

- In `src/groups/g1.rs#L165-168`, a reference to "expectation / convention used by Geth / Reth" should be included.
- In `src/groups/g2.rs#L123`, the code comment is misleading and should specify that the function must return a generator of $\mathbb{G}_2$, not the full twisted curve.
- In `src/hasher.rs#L6`, there are multiple instances in the code comments where `XMD` (Expand Message XMD) is implemented, while the comments incorrectly reference `XOF` (Expand Message XOF) instead.
- In `src/fields/fp6.rs#L3`, the module implements the cubic extension of the quadratic extension or the sextic extension of the base.
- In `src/groups/g2.rs#L121`, it should be specified that the generator of $\mathbb{G}_2$ is returned.

Mitigation

We recommend improving the comments as detailed above.

Status

The comments have been updated as suggested. More specifically:

- The `compute_naf` function has been removed, so the suggestion is no longer relevant.
- The rest of the comments (here, here, here and here) have been improved as recommended.

Verification

Resolved.

Suggestion 5: Correct Tracing Variable

Location

`src/groups/group.rs#L543`

Synopsis

In `src/groups/group.rs#L543`, three variables `t0`, `t1`, and `t2` should be traced. However, the third variable should be `t2` and not `t1` in order for the values to be traced correctly.

Mitigation

We recommend correcting the code as detailed above.

Status

The Warlock team has rectified the typographical error.

Verification

Resolved.

Suggestion 6: Add Note Clarifying Bugs in Reference Article

Location

[BGM+10]

Synopsis

In [BGM+10], Algorithm 17 contains two errors. Firstly, Step 6 should be: `t_5 ← a_1*a_2`. Secondly, Step 9 should be: `c_2 ← t_1 - t_4`. The code behaves correctly and is hence currently inconsistent with Algorithm 17 from the reference.

Mitigation

To avoid confusion and for better code maintenance, we recommend adding a note in the code comments regarding the inconsistencies present in [BGM+10], Algorithm 17.

Status

The Warlock team has updated the relevant comment in the codebase to highlight the typographical errors in the referenced article.

Verification

Resolved.

Suggestion 7: Rename Function To Match Functionality

Location

`src/BLS.sol#L121`

Synopsis

The function `isValidSignature` currently checks if the input is a point in the group $\mathbb{G}_1$. However, for a signature to be valid, the input also needs to satisfy the signature verification algorithm. Hence, it is confusing to name a function "`IsValidSignature`" if the intended behavior is to only check if the point is in $\mathbb{G}_1$. Otherwise, if the intended behavior is to actually check for a valid signature, then the code must add a check for signature verification.

Mitigation

We recommend adjusting the function `IsValidSignature` accordingly.

Status

This suggestion remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 8: Use Standard Terminology for Domain Separation Tag

Location

Multiple functions within `BLS.sol`.

Examples (non-exhaustive):

`src/BLS.sol#L95`

`src/BLS.sol#L244`

Synopsis

In some instances in the codebase, the domain separation tag is referred to as "`domain`" (e.g., in `hashToPoint`), while in other locations, it is sometimes referred to as "`dst`" (e.g., in `expandMsgTo96`). Standardizing on either `domain` or `dst` would facilitate easier understanding of the code and allow both maintainers and reviewers of the codebase to comprehensively understand the intended functionality of the implementation, which increases the likelihood for identifying potential errors that may lead to security vulnerabilities.

Mitigation

To improve code comprehensibility, we recommend being consistent in naming by using either `domain` or `dst`.

Status

This suggestion remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 9: Upgrade Solidity Version and Lock the Pragma

Location

`src/BLS.sol#L2`

`src/ModExp.sol#L2`

Synopsis

Smart contracts in the project have their pragma set to `>=0.8.23`. Compiling with different versions of the compiler might lead to unexpected results. In addition, older versions of the Solidity compiler may contain bugs that have been fixed in more recent versions of the compiler, including up-to-date security patches.

Mitigation

In order to maintain consistency and to prevent unexpected behavior, we recommend that the Solidity compiler version be pinned by removing ">=" and using the latest version of the Solidity compiler.

Status

This suggestion remains unresolved at the time of verification.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing, along with the use of tools, including AI, to support our code review efforts. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.