Mysten Fastcrypto BLS12381 Group Audit

Shresth Agrawal[1,2] Pyrros Chaidos[1,3]

[1] Common Prefix
[2] Technical University of Munich
[3] University of Athens

November 15, 2023
Last update: January 18, 2024

1 Overview

1.1 Introduction

Mysten Labs commissioned Common Prefix to audit the BLS-381 group implementation within their fastcrypto library. The primary objectives of the audit were to assess the security, adherence to the relevant publications, and also investigate performance optimizations, and code quality improvements to these particular implementations. Fastcrypto is a Rust-based library that implements selected cryptographic primitives and also serves as a wrapper for several carefully chosen cryptography crates, ensuring optimal performance and security for Mysten Labs' software solutions, including their blockchain platform, Sui.

This audit report evaluates the BLS-381 group implementation within the fastcrypto library. We have audited the code for security, efficiency, and reliability. The scope of this audit was limited to the fastcrypto implementation and did not extend to the library's dependencies or any downstream applications.

1.2 Audited Files

1. [a63b6996] fastcrypto/src/groups/bls12381.rs

1.3 Disclaimer

This audit does not give any warranties on the bug-free status of the given code, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

The scope of the audit was constrained exclusively to the Fastcrypto wrapper code, with no examination conducted on its associated dependencies. Furthermore, the audit does not encompass any reference string generation functionality in terms of code or execution.

1.4 Executive Summary

Overall, the BLS-381 group implementation is of very high quality and adhering to Rust's best practices.

For most complex group operations the code relies on the upstream BLST library for optimized implementations. The code implements some custom scalar-element multiplications and serializations.

Our findings mostly concern unclear semantics and documentation of the `blst_fr` type, the non unique byte representation of FP12 elements, and code refactoring for safer lifetimes in unsafe type conversions.

1.5 Findings Severity Breakdown

The findings are classified under the following severity categories according to the impact and the likelihood of an attack.

Level	**Description**
High	Logical errors or implementation bugs that are easily exploited. In the case of contracts, such issues can lead to any kind of loss of funds.
Medium	Issues that may break the intended logic, are deviations from the specification, or can lead to DoS attacks.
Low	Issues harder to exploit (exploitable with low probability), can lead to poor performance, clumsy logic, or seriously error-prone implementation.
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable, and easier to maintain.

2 Findings

2.1 High

None Found.

2.2 Medium

M01: Non unique deserialization of `GTElement`.

Affected Code: src/groups/bls12381.rs (line 539)

Summary: `GTElement::from_byte_array` allows for multiple byte representations to deserialize to the same element. This is caused due to the use of `blst_fp_from_bendian` which performs a mod reduction for byte array with value greater than p. This can lead to undefined behaviour for the higher level usage of the library which expects unique byte representation.

Suggestion: In the inner most loop (line 550) check that each byte array is in cannonical representation of the FP element and return an error if not. This can be done by checking that the byte array is less than p or by creating a cannonical big endian representation (using `blst_bendian_from_fp`) of deserialized element and comparing it to the input byte array.

Status: Resolved [5afe77d3 4dfb26f5]

2.3 Low

None Found.

2.4 Informational

I-01: Undefined lifetime of blst point slices in `multi_scalar_mul`.

Affected Code: src/groups/bls12381.rs (lines 157,345)

Summary: The `multi_scalar_mul` implementations for both `G1Element` and `G2Element` perform unsafe type conversion from `&[Self]` to `&[blst_p1]` or `&[blst_p2]` respectively. The lifetime of the output slice is not well defined and is inferred from its usage. It is advisable to tie the lifetime of the input slice to the output slice.

Suggestion: A potential way to do this is by wrapping the unsafe component into a seperate function. The rust compiler implicitly ties the lifetime of the input parameters to the output for such function.

```rust
fn from_blst_p1_slice(points: &[G1Element]) -> &[blst_p1] {
    // SAFETY: the cast from `&[G1Element]` to `&[blst_p1]` is safe because
    // G1Element is a transparent wrapper around blst_p1. The lifetime of
    // output slice is the same as the input slice.
    unsafe { std::slice::from_raw_parts(points.as_ptr() as *const blst_p1, points.len()) }
}
```

Status: Resolved [fd811174, c50c9f07]

I-02: Scalar type is sparsely documented.

Affected Code: src/groups/bls12381.rs (lines 54,747)

Summary: The `Scalar` type, implemented as `blst_fr` is not adequately documented. The underlying blst codebase provides two representation for scalars, `blst_fr` and `blst_scalar`. The blst library uses `blst_scalar` for all the frontend operations (e.g. using it for bls private key) while `blst_fr` is used for all the low level operations and arithmetic. The fastcrypto implementation uses `blst_fr` as the `Scalar` type but also performs several type conversions from/to `blst_scalar`. The codebase doesn't document why one type is used over the other at several places. Another instance of this can be seen in the "magic" value of `BLST_FR_ONE`. Whilst the code is correct, this leaves the potential for future maintainability issues and may also impact downstream development.

Suggestion: Add documentation for the usage of `blst_fr`. The documentation provided by the blst codebase is very sparse. Chapter 14 Paragraph 14.3.2 of [MVOV18] is a good reference which provides the main motivation for using Montgomery forms on which `blst_fr` is based. We note that some of the type choices are restricted by methods exposed by the blst codebase which eliminates the idea of potential refactor which only uses one of the two scalar types.

Status: Acknowledged

I-03: Multiple type conversions in `GTElement` `mul`.

Affected Code: src/groups/bls12381.rs (line 480)

Summary: Currently the `mul` implementation for `GTElement` performs all the scalar arithmetic in `blst_fr`. This increases the complexity of the implementation (requires several type conversions to `blst_scalar` and usage of unsafe functions). The implementation is simplified if the scalar arithmetic is performed in `blst_scalar` instead.

Suggestion: Below is a reference `mul` implementation using `blst_scalar`.

```
fn mul(self, rhs: Scalar) -> Self {
    let mut n = blst_scalar::default();
    unsafe{
        blst_scalar_from_fr(&mut n, &rhs.0);
    }

    let bytes_len = size_in_bytes(&n);
    let bits_len = size_in_bits(&n, bytes_len);
```

```rust

        if bits_len == 0 {
            return Self::zero();
        }
        if bits_len == 1 {
            return self;
        }

        let mut y: blst_fp12 = blst_fp12::default();
        let mut x = self.0;

        for i in 0..(bits_len - 1) {
            // Get the bit at the ith position.
            if n.b[i / 8] & (1 << (i % 8)) == 1 {
                y *= x;
            }
            unsafe {
                blst_fp12_sqr(&mut x, &x);
            }
        }
        y *= x;
        Self::from(y)
    }
```

We note that the above code doesn't effect the performance of the implementation as the bottleneck of the operation is the FP12 multiplications.

Status: Acknowledged

References

MVOV18. Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 2018.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.