Mysten Fastcrypto BLS12381 Audit

Shresth Agrawal[1,2] Petros Angelatos[1,3]
Pyrros Chaidos[1,4]

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

April 07, 2023
Last update: May 27, 2023

1 Overview

1.1 Introduction

Mysten Labs commissioned Common Prefix to conduct an audit of the BLS12381 [4] implementation within their fastcrypto library. The primary objectives of the audit were to assess the security and adherence to relevant standards and investigate performance optimizations and code quality improvements for this specific implementation. 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 [7, 10]

BLS [2,3,5] signatures are known for their efficient aggregation properties, which can significantly reduce the size of signature sets in blockchain protocols and improve overall scalability. The fastcrypto BLS12381 implementation primarily serves as a wrapper around the blst library, which is written in C and Assembly. The blst library has been extensively audited [9] and is currently used in production by several major blockchains, such as Ethereum [6, 8, 12].

This audit report thoroughly evaluates the usage of the BLS12381 implementation within the fastcrypto library, specifically in the context of the Sui blockchain, as opposed to general-purpose usage (Cf. Section 3.1). The audit findings are categorized by severity, and we provide proposed solutions for each identified issue. Our evaluation primarily focuses on the code's security, efficiency, and reliability. It's important to note that this audit is limited in scope to the code associated with non-experimental features and does not encompass the library's dependencies or other components.

1.2 Audited Files

1. [120b30ca] fastcrypto/src/bls12381/mod.rs (excluding experimental code between lines 208 - 272).

1.3 Disclaimer

This audit does not give any warranties on the bug-free status of the given code. 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.

1.4 Executive Summary

Overall the BLS12381 implementation within the fastcrypto library is a well-crafted, secure, and optimized wrapper around the blst library. The blst library itself is heavily audited [9] and battle-tested in publicly deployed blockchain systems e.g., Ethereum [8]. Throughout the audit process, we identified several issues, all of which were either low or informational in severity.

Notably, the implementation of the `verify_different_msg` function (L-06) does not intrinsically enforce the requirement for message uniqueness as stipulated by the RFC specification. This can enable a rogue key attack [2]. Similarly, the `verify` function (L-07) for a common message does not list or enforce any particular requirements on the keys. Both of these issues are of low severity as Sui uses Proofs of Possession, which mitigates both problems.

Additionally, all the verification functions presuppose a pre-validation of the public key (L-01). While such a strategy aligns with the implementation specifics of the Narwhal protocol, it might not be immediately apparent to users in a broader context, thus potentially leading to misuse.

Discrepancies in the implementation of equality and ordering traits for `BLS12381PublicKey`, `BLS12381PrivateKey`, `BLS12381Signature`, and `BLS12381AggregateSignature` were also observed. While these discrepancies do not pose a direct security threat, they may yield unexpected results when consistency between these traits is assumed.

In summary, the fastcrypto BLS12381 implementations have been found to be of high quality. Some of the issues mentioned above might lead to more severe problems if the library is used in a context outside

the Narwhal consensus protocol [7] and the Sui blockchain [10]. However, these issues can be easily addressed.

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, is a deviation from the specification, or can lead to DoS attacks.
Low	Issues harder to exploit (exploitable with low probability), issues that 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

None found.

2.3 Low

L-01: Overflow in `get_random_scalar` calculation

Affected Code: fastcrypto/src/bls12381/mod.rs (line 286)

Summary: The current implementation creates an overflow when `BLS_BATCH_RANDOM_SCALAR_LENGTH` is set to 128. This occurs because the existing calculation involves a left shift operation on a 64-bit integer requiring a 65th-bit position.

Suggestion: Replace the current calculation with one that uses a larger numeric type to prevent overflow.

Suggested Fix:

```
--- a/fastcrypto/src/bls12381/mod.rs
+++ b/fastcrypto/src/bls12381/mod.rs
@@ -283,7 +283,7 @@ fn get_random_scalar<Rng:
    AllowedRng>(rng: &mut Rng) -> blst_scalar {
         vals[1] = rng.next_u64();

         // Reject zero as it is used for
            multiplication.
-        let vals1_lsb = vals[1] & ((1 <<
    (BLS_BATCH_RANDOM_SCALAR_LENGTH - 64)) - 1);
+        let vals1_lsb = vals[1] & (((1u128 <<
    (BLS_BATCH_RANDOM_SCALAR_LENGTH - 64)) - 1) as u64);
         if vals[0] | vals1_lsb != 0 {
             break;
         }
```

This fix performs the bit shift operation in a `u128` type and then casts the result back to `u64`.

Status: Open

L-02: `get_random_scalars` function returns an array of size 1 when `n` is set to 0

Affected Code: fastcrypto/src/bls12381/mod.rs (line 309)

Summary: The `get_random_scalars` function returns an array of size one even if the input parameter `n` is set to 0. Although the existing code cannot be exploited, it could potentially lead to unintended behavior in future usage of the function.

Suggestion: Consider adding a check for `n` being 0, and panicing with an error message. Alternatively, modify the `get_random_scalars` function to return an empty array when `n` is set to 0, ensuring that the output array size matches the input parameter. This will help prevent any potential future misuse of the function.

Status: Open

L-03: Public Key Validation in Verification Functions

Affected Code:

- fastcrypto/src/bls12381/mod.rs (line 196)
- fastcrypto/src/bls12381/mod.rs (line 641)
- fastcrypto/src/bls12381/mod.rs (line 661)
- fastcrypto/src/bls12381/mod.rs (line 682)

Summary: The current implementation of `BLS12381PublicKey.verify`, `BLS12381AggregateSignature.verify`, `BLS12381AggregateSignature.verify_different_msg`, and `BLS12381AggregateSignature.batch_verify` do not validate the public key and expect it to be validated before the functions are called. A `TODO` comment in the code mentions that the current implementation of Narwhal [7] (the consensus protocol of the Sui blockchain [10]) uses several serialization/deserialization of validated public keys and that a refactor is planned to enable validation during deserialization. This can lead to misuse of these functions by fastcrypto users other than the Narwhal consensus implementations itself.

Suggestion: Until the refactor is done, we suggest introducing a new type `UnvalidatedBLS12381PublicKey` which wraps `BLS12381PublicKey`. `UnvalidatedBLS12381PublicKey` should have an `assume_valid` function which returns an unvalidated `BLS12381PublicKey`, and an `into_valid` function, which returns a validated `BLS12381PublicKey`. By default, the deserialization of `BLS12381PublicKey` should always be validated. Here is a rough code snippet for the above suggestion:

```
struct UnvalidatedBLS12381PublicKey(BLS12381PublicKey);

impl UnvalidatedBLS12381PublicKey {
  /// Asume that this key is validated without
     checking.
  pub fn assume_valid(self) -> BLS12381PublicKey {
     self.0
  }

  pub fn into_valid(self) -> Result<BLS12381PublicKey,
     FastCryptoError> {
      self.0.pubkey.validate().map_err(|_e|
         FastCryptoError::InvalidInput)?;
      self.0
   }
}

// TODO: add fast deserialization of
   UnvalidatedBLS12381PublicKey which doesn't validate
// TODO: update the deserialization of
   BLS12381PublicKey to always validate.
```

Status: Open

L-04: Inconsistency between `PartialEq` trait implementations for different BLS12381 structures

Affected Code:

- fastcrypto/src/bls12381/mod.rs (line 102)
- fastcrypto/src/bls12381/mod.rs (line 322)
- fastcrypto/src/bls12381/mod.rs (line 390)
- fastcrypto/src/bls12381/mod.rs (line 545)

Summary: The current implementation of `PartialEq` trait for different BLS12381 structures are inconsistent. `BLS12381PublicKey` and `BLS12381AggregateSignature` use the blst implementation for `PartialEq`, while `BLS12381PrivateKey` and `BLS12381Signature` first convert to byte representation and then compare. This is especially problematic for `BLS12381AggregateSignature` and `BLS12381Signature`, which have different implementations for `PartialEq` even though they wrap the same blst object.

Suggestion: Use the blst implementation of `PartialEq` for all structures except `PrivateKey`, as blst does not have a `PartialEq` implementation for it.

Status: Open

L-05: Inconsistency between `PartialEq`, `PartialOrd`, and `Ord` trait implementations for `BLS12381PublicKey`

Affected Code:

- fastcrypto/src/bls12381/mod.rs (line 102)
- fastcrypto/src/bls12381/mod.rs (line 110)
- fastcrypto/src/bls12381/mod.rs (line 117)

Summary: The `PartialOrd` and `Ord` trait implementations for `BLS12381PublicKey` are inconsistent with the `PartialEq` implementation, as the former converts the underlying `blst::PublicKey` to bytes and perform byte-level comparison, while the latter directly uses the underlying blst implementation. The Rust documentation explicitly states that implementations of `PartialOrd` and `Ord` must be consistent with `PartialEq`.

Suggestion: Do not provide an implementation for the `PartialOrd` and `Ord` traits from the library to avoid any unexpected behavior. If needed, a byte-level comparison can be implemented by the application itself.

Status: Open

L-06: `verify_different_msg` function should check for message uniqueness

Affected Code: fastcrypto/src/bls12381/mod.rs (line 661)

Summary: The current implementation of the `verify_different_msg` function does not check for the uniqueness of the messages, as required by the RFC section 3.1.1 (AggregateVerify) [4]. The function relies on the `blst::Signature.aggregate_verify` function, which does not perform this check, as indicated by a "`TODO`" comment in the blst library. This can lead to rogue key attacks [2]. However, this is not a problem in the context of Sui blockchain as Proofs of Possession (PoP) is used.

Suggestion: Update the `verify_different_msg` function to ensure that all messages passed to it are unique in accordance with the RFC specification.

Suggested Fix:

```
--- a/fastcrypto/src/bls12381/mod.rs
+++ b/fastcrypto/src/bls12381/mod.rs
@@ -30,6 +30,7 @@ use fastcrypto_derive::{SilentDebug,
    SilentDisplay};
 use once_cell::sync::OnceCell;
 use std::{
     borrow::Borrow,
+    collections::HashSet,
     fmt::{self, Debug, Display},
     mem::MaybeUninit,
     str::FromStr,
@@ -663,6 +664,11 @@ impl AggregateAuthenticator for
    BLS12381AggregateSignature {
         pks: &[<Self::Sig as Authenticator>::PubKey],
         messages: &[&[u8]],
     ) -> Result<(), FastCryptoError> {
+        let mut unique_msgs: HashSet<&[u8]> =
    HashSet::with_capacity(messages.len());
+        unique_msgs.extend(messages.iter());
+        if unique_msgs.len() != messages.len() {
+            return Err(FastCryptoError::InvalidInput);
+        }
         // Validate signatures but not public keys
             which the user must validate before calling
             this.
```

```
let result = self
    .sig
```

Status: Open

L-07: `verify` function should specify semantics and security assumptions

Affected Code: fastcrypto/src/bls12381/mod.rs (line 641)

Summary: The current implementation of the `verify` function uses blst's `fast_aggregate_verify` to perform the actual verification. This can be unsafe unless the public keys are known to be honestly generated. In the proposed draft [4], `fast_aggregate_verify` is only described for BLS augmented with Proofs of Possession (PoP) which explicitly harden BLS against rogue key attacks. Without PoPs or a similar assumption, an adversary can produce aggregate signatures for a set containing the keys of honest users without their knowledge or cooperation by using adversarial (rogue) public keys designed to attack the users in mind. This is not a problem in the context of Sui blockchain as PoPs are used.

Suggestion: Whilst explicit proofs of possession are not the only means to counteract rogue key attacks[5], it needs to be made clear that (1) the function is only safe to use when specific assumptions are met and (2) the relevant assumptions should be made explicit.

As additional precautions against unsafe use of the primitives by future developers we also suggest

- Changing the function name to draw caution to the additional requirements.
- Refactoring the code so that the types of aggregates of signatures produced by keys that are not known to be hardened against rogue key attacks are not valid arguments.

Status: Open

2.4 Informational

I-01: Simplify usage of `OneCell` with `get_or_init` instead of `get_or_try_init`

Affected Code:

- fastcrypto/src/bls12381/mod.rs (line 137)
- fastcrypto/src/bls12381/mod.rs (line 348)

[5] cf. the MSKR research paper [1], or implicit PoPs via an application-specific registration process

- fastcrypto/src/bls12381/mod.rs (line 407)
- fastcrypto/src/bls12381/mod.rs (line 568)

Summary: The current implementation uses `get_or_try_init` with `OneCell`, which expects that the initialization function passed to it might return a `Result.Err`. However, in all calls to `get_or_try_init`, an explicit `Ok(something)` is returned. This can be simplified by using `get_or_init` instead and not wrapping the object in `Ok`.

Suggestion: Consider replacing `get_or_try_init` with `get_or_init` in the affected code to simplify the calls and remove the unnecessary wrapping of the object in `Ok`.

Suggested Fix: Here is an example fix

```
--- a/fastcrypto/src/bls12381/mod.rs
+++ b/fastcrypto/src/bls12381/mod.rs
@@ -345,8 +345,7 @@ impl Drop for BLS12381PrivateKey {
 impl AsRef<[u8]> for BLS12381PrivateKey {
     fn as_ref(&self) -> &[u8] {
         self.bytes
-            .get_or_try_init::<_, eyre::Report>(||
    Ok(self.privkey.to_bytes()))
-            .expect("OnceCell invariant violated")
+            .get_or_init(|| self.privkey.to_bytes())
     }
 }
```

Status: Open

3 Supplementary Information

3.1 Usage Outside the Context of the Sui Blockchain

Usage of the BLS12381 implementation outside the context of the Sui blockchain should be done carefully. While some of the previously discussed issues were categorized as low severity within the Sui blockchain due to specific security measures and design choices, they may pose higher severity risks in other environments. Specifically, the following points should be taken into account:

The issue L-06 pertains to the lack of enforcement of message uniqueness in the `verify_different_msg` function, which is currently considered low severity within the Sui blockchain because of the use of proofs of possession. However, when utilizing the library outside the Sui blockchain, it becomes crucial to ensure message uniqueness to prevent potential rogue key attacks.

Similarly, regarding issue L-07, which highlights the lack of specified semantics and security assumptions in the `verify` function, it is recommended to use the `verify` function only with proof of possession or a similar assumption on the public key.

Another issue, L-03, relates to the lack of public key validation on deserialization. While this is currently managed within the Sui blockchain by handling invalidated keys carefully. Proper public key validation on deserialization should be performed if the library is used outside the context of Sui.

Reviewing all the issues mentioned in this report and assessing their relevance and impact on your specific use case when using the library outside the Sui blockchain is generally recommended.

In conclusion, while the fastcrypto BLS12381 implementation has been found to be generally safe to use, caution should be exercised when employing the library outside the context of the Sui blockchain.

3.2 Zero Splitting Attack

A related notion to rogue key attacks is that of "zero-splitting" [11] attacks where a number of adversarial keys are set so as to sum to zero, which in turn makes their aggregate signatures exhibit non-standard properties, such as being valid for multiple different messages. This does not contradict security definitions describing forgeries but may prove problematic for higher-level protocols. PoPs and standard aggregation with different messages are explicitly not sufficient to prevent them, as checking for keys with a particular sum is expensive to do preemptively. The approach of MSKR [1] may be more fruitful but lies outside the scope of this audit.

4 Acknowledgement

The authors of the report would like to thank Dionysis Zindros for giving helpful feedback on the report.

References

1. F. Baldimtsi, K. K. Chalkias, F. Garillot, J. Lindstrom, B. Riva, A. Roy, A. Sonnino, P. Waiwitlikhit, and J. Wang. Subset-optimized bls multi-signature with key aggregation. Cryptology ePrint Archive, Paper 2023/498, 2023. `https://eprint.iacr.org/2023/498`.

2. D. Boneh, M. Drijvers, and G. Neven. Compact multi-signatures for smaller blockchains. In T. Peyrin and S. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 435–464, Cham, 2018. Springer International Publishing.
3. D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. pages 416–432, 2003.
4. D. Boneh, S. Gorbunov, R. S. Wahby, H. Wee, C. A. Wood, and Z. Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-05, Internet Engineering Task Force, June 2022. Work in Progress.
5. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. 17(4):297–319, Sept. 2004.
6. V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
7. G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. pages 34–50, 03 2022.
8. E. Foundation. Ethereum consensus specifications.
9. N. Group. Blst cryptographic implementation review, January 2021.
10. M. Labs. The sui smart contract platform.
11. N. T. M. Quan. 0. Cryptology ePrint Archive, Paper 2021/323, 2021. `https://eprint.iacr.org/2021/323`.
12. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

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.