Mysten Fastcrypto ECDSA Secp256r1 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

June 08, 2023
Last update: September 11, 2023

1 Overview

1.1 Introduction

Mysten Labs commissioned Common Prefix to audit the ECDSA secp256r1 implementation within their fastcrypto library. The primary objectives of the audit were to assess the security, adherence to the relevant RFCs, 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.

ECDSA (Elliptic Curve Digital Signature Algorithm) is a cryptographic algorithm utilizing elliptic curves to generate digital signatures [1]. It offers recoverable signatures, which allow for public key recovery from the signature itself. The secp256r1 is a standard elliptic curve with prime order of size 256 bits. The curve's parameters were generated using verifiable randomness to ensure security against any special purpose attacks or trapdoors [2].

Fastcrypto's ECDSA secp256r1 implementation employs optimized curve multiplication and multimultiplication techniques using precomputation. This approach enhances processing speed, albeit with a larger memory footprint. This is crucial as these operations account for the majority of computation time of the verifier. The implementation relies on p256 crate and several Arksworks crates, including ark-secp256r1, ark-ff, ark-ec, etc., for the underlying curve point, scalar and field element operations.

This audit report comprehensively evaluates the ECDSA secp256r1 implementation within the fastcrypto library. The findings are categorized by severity, accompanied by proposed solutions for each identified issue. We have audited the code for security, efficiency, and reliability. The scope of this audit was limited to the ECDSA secp256r1 and its supporting fast multiplication implementations within the Fastcrypto library and did not extend to the library's dependencies or other components.

1.2 Audited Files

1. [963205c6] fastcrypto/src/secp256r1/mod.rs
2. [963205c6] fastcrypto/src/secp256r1/recoverable.rs
3. [963205c6] fastcrypto/src/secp256r1/conversion.rs
4. [963205c6] fastcrypto/src/groups/secp256r1.rs
5. [963205c6] fastcrypto/src/groups/multiplier/integer_utils.rs
6. [963205c6] fastcrypto/src/groups/multiplier/windowed.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.

1.4 Executive Summary

Overall, both the ECDSA secp256r1 and the corresponding fast multiplication implementation are of very high quality, following the RFC guidelines and adhering to Rust's best practices.

The implementation cherry-picks and stitches components from two existing implementations of ECDSA secp256r1, namely ark-secp256r1 and p256. While such an approach provides the benefit of leveraging established components, it concurrently amplifies complexity and widens the potential for errors. An instance of this is evidenced in the `fq_arkworks_to_p256` method, intended to convert arkworks field element to p256 field element. However, the function inadvertently returns a p256 scalar instead of the desired p256 field element. The difference in orders of fields would lead to unexpected value modification during the type conversion.

We identified two primary concerns regarding the `Scalar` implementation. First, the `rand` function doesn't generate scalars uniformly random. Second, the deserializing method allows for multiple byte arrays to deserialize to the same `scalar`.

Even though it is a good practice for crypto libraries to clear the memory storing the secret after it is used, doing so incorrectly can lead to leakage. During the audit, we identified that the bytes of the private key were leaked on the stack during the cleanup operation itself.

We also observed two deviations from the protocol standards. First, the signing function for recoverable signatures did not implement the code path to generate signatures with IDs 2 and 3. Second, the message hash passed to the `generate_k` function is not reduced modulo the group order, as required by the RFC6979 [3].

Beyond these primary concerns, our audit also pinpointed a series of minor discrepancies, such as overlooked base cases, arithmetic overflows in utility functions, and missing checks. We also suggested minor code refactoring to improve the overall code quality. In conclusion, the current iteration of the code demonstrates substantial quality, indicative of meticulous design and development. The issues presented in the report are minor and can be addressed easily.

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

M-01: Scalars from `rand` are not uniformly random

Affected Code: src/groups/secp256r1.rs (line 98)
Summary: The `rand` function first generates 32 bytes uniformly at random and then uses `from_be_bytes_mod_order` to convert the bytes to a valid scalar. The mod operation reduces the resulting integer if it is greater than the order, skewing the generated scalars' probability.
Suggestion: Using more uniform bytes (e.g. twice the size of scalars) would make the bias negligible. Alternatively, rejection sampling could be used where bytes greater than the order are discarded and new ones are sampled until valid ones are found.
Status: Resolved [10215f09]

M-02: Small scalars have multiple representations

Affected Code: src/groups/secp256r1.rs (line 112)
Summary: The `from_byte_array` function uses mod order (`from_le_bytes_mod_order`) to convert from bytes to scalars, allowing for multiple byte representations for small scalars. This may allow trivial malleability attacks in downstream applications.
Suggestion: Enforce a canonical representation by checking that the modular reduction is idempotent. This ensures that high byte values cannot be used as aliases for their reduced equivalent.
Status: Resolved [c53399a2, d108330a3]

M-03: Missing code path for recoverable signatures

Affected Code: src/secp256r1/recoverable.rs (line 167)
Summary: `sign_recoverable_with_hash` does not implement the code path to generate signatures with `recovery_id` 2 and 3.
Suggestion: The code should check if `big_r.x()` is high, i.e. r has been reduced in the resulting signature. If so, set the high bit of the `recovery_id`.
Status: Resolved [d695bba3]

M-04: Potential private key leakage via `drop`

Affected Code: src/secp256r1/mod.rs (line 308)

Summary: `drop` implementation leaks private keys on the stack. The `Secp256r1PrivateKey` implementation uses `OnceCell` to cache the byte representation of the private key and later calls `bytes.take().zeroize()` within the zeroize trait implementation to wipe the memory. The `.take()` function uses `move` to transfer the ownership. The `move` operation compiles to `memcopy`, which copies the bytes to the destination of the move, subsequently making the initial location inaccessible (but still containing the original bit pattern). The `drop` implementation does not run for the original location and there is no way to customize the move behavior in `Rust`. This is a documented limitation of the zeroize crate.

Suggestion: There are several places where the move operation is applied on the private keys, which would all leak the secret on the stack. One practical approach to protect against leaking secret memory is to put the secret bytes in a `Box` as soon as possible while being extremely careful about moves until that point. Once the secret data gets inside, the `Box` moves are safe because only the pointer to the box will be copied around, not the actual data.

Status: The suggested approach requires a significant refactoring of the codebase; therefore, the team has decided only to perform a best-effort fix for the problem. This is done by wrapping the `bytes` into `Zeroizing` wrapper instead of performing `bytes.take().zeroize()` inside the `drop`. The `Zeroizing` wrapper carefully zeroizes the bytes without moving. This ensures that the secret bytes are not leaked during the `drop`.
Fixes: [45275491, e884cc09]

M-05: Scalar type used to represent field element

Affected Code: src/secp256r1/conversion.rs (line 32)

Summary: The function `fq_arkworks_to_p256` is expected to convert an arkworks field element (`ark_secp256r1::Fq`) to a p256 field element (`p256::FieldElement`). However, the function returns a p256 scalar (`ark_secp256r1::Scalar`) instead of the desired p256 field element. This is erroneous as these types represent different fields, and the conversion is not performed with the intention to move from one field to the other explicitly. Additionally, the issue is rare enough to evade randomized testing.

Suggestion: The function should return `p256::FieldElement` type instead. Add a test case with a scalar of size greater than the order of the field to ensure that the fix is correct.
Status: Missing test case. Resolved [2c00745e]

2.3 Low

L-01: Windowed multiplication fails for cache size 1

Affected Code: src/groups/multiplier/windowed.rs (line 56)
Summary: `new` doesn't handle the base case when the cache size is 1 which is a valid power of 2. For a cache size of 1, `cache[1]` will be out of bounds.
Suggestion: Add check for the case when `CACHE_SIZE` is 1.
Status: Resolved [614b397f]

L-02: Missing check for `CACHE_SIZE` being an exact power of 2

Affected Code: src/groups/multiplier/windowed.rs (lines 59,95)
Summary: If `CACHE_SIZE` is not an exact power of 2, this leads to undefined behavior in the above-mentioned lines of code.
Suggestion: Check that `CACHE_SIZE` is an exact power of 2. Alternatively, set `CACHE_SIZE` to be `2**SLIDING_WINDOW_WIDTH`.
Status: Resolved [b51d9838]

L-03: `mul` is not constant time due to upstream implementation

Affected Code: src/groups/multiplier/windowed.rs (line 65)
Summary: While the method used is constant time, `mul` is not constant time as the `arkwork` group's addition implementation is also not constant time.
Suggestion: Adjust documentation to reflect upstream dependency.
Status: Open

L-04: `compute_multiples` is incorrect when `window_size` is 1

Affected Code: src/groups/multiplier/windowed.rs (line 213)
Summary: `compute_multiples` returns incorrect results when `window_size` is 1. The value is strictly positive and thus passes the assert statement. The correct value of `smallest_multiple` in that case should equal `base_element` rather than `base_element.double()`.
Suggestion: Initialize `smallest_multiple` to `base_element` (instead of `base_element.double()`), and change the `for` loop to start at 1 (instead of 2).
Status: Resolved [614b397f]

L-05: Overflow can cause arbitrary read of bits outside the given range

Affected Code: src/groups/multiplier/integer_utils.rs (line 31)
Summary: The function `get_lendian_from_substring` is expected to return the little-endian representation of the substring of a given byte and a range $[start, end)$. If $end - start$ is greater than 8 then the following code `((1 << (end - start))- 1)as u8` causes an overflow leading to an arbitrary read of bits outside the given range.
Suggestion: Add code to handle the case where `end` is greater than 8.
Status: Resolved [a05e5806]

L-06: Potential overflow in `div_ceil`

Affected Code: src/groups/multiplier/integer_utils.rs (line 37)
Summary: The `div_ceil` implementation uses `(numerator + denominator - 1)/ denominator` to perform the ceiling division. This can overflow if the values of either the `numerator` or `denominator` are big enough.
Suggestion: Refactor `(numerator + denominator - 1)/ denominator` to `1 + ((numerator - 1)/ denominator)` and add a check that `numerator` is greater than zero.
Status: Resolved [066a98e0]

L-07: `from_bytes` should check that `recovery_id` is in the correct range

Affected Code: src/secp256r1/recoverable.rs (line 71)
Summary: `from_bytes` does not check that `recovery_id` is in range [0, 3].
Suggestion: Add assertion that `recovery_id < 4`.
Status: Resolved [1805fec7]

L-08: `zeroize` implementation for `Secp256r1PrivateKey` is superfluous

Affected Code: src/secp256r1/mod.rs (line 301)
Summary: The `Secp256r1PrivateKey` implements both `zeroize` and `drop` traits with the exact same implementation. This makes the `zeroize` trait implementation useless as the `drop` function will always be called.
Suggestion: Either make the `drop` call the `zeroize` function or just remove the implementation.
Status: Resolved [9a3a179f]

L-09: k value generation diverges from RFC6979

Affected Code: src/secp256r1/mod.rs (line 402)
Summary: The value `H::digest(msg).digest` passed to the `rfc6979::generate_k` function should be reduced modulo the group order as required by the function documentation and RFC. Diverging from this is safe to do (Sect 3.6 in RFC6979) but we believe there is little benefit to be had.
Suggestion: Reduce `z` modulo the group order before calling `rfc6979::generate_k`. Alternatively, flag the divergence in the documentation.
Status: Resolved [8d1b4e9f, d97faf43]

2.4 Informational

I-01: Inconsistency between `PartialEq`, `PartialOrd`, and `Ord` trait implementations for `Secp256r1PublicKey`

Affected Code: src/secp256r1/mod.rs (lines 122,128,134)
Summary: The `PartialOrd` and `Ord` trait implementations for `Secp256r1PublicKey` are inconsistent with the `PartialEq` implementation, as the former converts the underlying `p256::ecdsa::VerifyingKey` to bytes and performs byte-level comparison, while the latter directly uses the upstream implementation of the trait. The Rust documentation explicitly states that implementations of `PartialOrd` and `Ord` must be consistent with `PartialEq`.
Suggestion: Use the upsteam implementation for the `PartialOrd` and `Ord` traits as well.
Status: Resolved [b67d65bb]

I-02: Use member type for better code consistency

Affected Code: src/secp256r1/mod.rs (lines 153,159)
Suggestion: Use `<ProjectivePoint as GroupElement>::ScalarType` instead of `crate::groups::secp256r1::Scalar` for better code consistency.
Status: Resolved [664d853d]

I-03: Simplify usage of `OnceCell` with `get_or_init` instead of `get_or_try_init`

Affected Code:

- src/secp256r1/mod.rs (lines 229,296,348)
- src/secp256r1/recoverable.rs (line 92)

Summary: The current implementation uses `get_or_try_init` with `OnceCell`, 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`.

Status: Resolved [e884cc09]

I-04: Use `ark_secp256r1::Projective` to reduce conversions

Affected Code:

- src/secp256r1/conversion.rs (line 40)
- src/secp256r1/recoverable.rs (lines 222,235)
- src/secp256r1/mod.rs (lines 196,210)

Summary: The output value of `affine_pt_p256_to_arkworks` is converted to `ark_secp256r1::Projective` in all of its uses.

Suggestion: Refactor `affine_pt_p256_to_arkworks` to directly return `ark_secp256r1::Projective` and remove the follow-up conversions.

Status: Resolved [acfb776f]

References

1. Certicom research, sec 1: Elliptic curve cryptography, 2010. `https://www.secg.org/sec1-v2.pdf`.
2. Certicom research, sec 2: Recommended elliptic curve domain parameters, 2010. `https://www.secg.org/sec2-v2.pdf`.
3. T. Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, Aug. 2013.

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.