

**Hardening Blockchain Security with Formal Methods** 

## **FOR**



**Smooth Crypto Library** 

Sponsored By:



Linea



## ► Prepared For:

Smoo.th

https://github.com/get-smooth/crypto-lib

## ► Prepared By:

Ajinkya D. Rajput Evgeniy Shishkin

#### ► Contact Us:

contact@veridise.com

## **▶** Version History:

Sep. 13, 2024 V3 Sep. 06, 2024 V2 Sep. 05, 2024 V1

Jul. 29, 2024 Initial Draft

## © 2024 Veridise Inc. All Rights Reserved.

# **Contents**

Co	ontents	iii
1	Executive Summary	1
2	Project Dashboard	5
3	Audit Goals and Scope	7
	3.1 Audit Goals	7
	3.2 Security Assessment Methodology & Scope	7
	3.3 Classification of Vulnerabilities	8
4	Vulnerability Report	9
	4.1 Detailed Description of Issues	10
	4.1.1 V-SCL-VUL-001: Incorrect fallback implementation for RIP7696	10
	4.1.2 V-SCL-VUL-002: EdDSA verification is vulnerable to malleability attacks	12
	4.1.3 V-SCL-VUL-003: Function takes secret key as argument	13
	4.1.4 V-SCL-VUL-004: Maintainability observations	14
	4.1.5 V-SCL-VUL-005: Unspecified revert messages	16
	4.1.6 V-SCL-VUL-006: Gas optimizations	17



From Jul. 18, 2024 to Jul. 30, 2024, Smoo.th engaged Veridise to conduct a security review of their Smooth Crypto Library. The security review covered the implementation of ECDSA signature generation and verification routines over the P256 curve, as well as EdDSA signature generation and verification routines over the Ed25519 elliptic curve. Veridise conducted the security assessment over 18 person-days, with security analysts reviewing the code over 9 days on commit c559657. The security review process involved a tool-assisted analysis of the source code performed by Veridise security analysts as well as thorough code review.

**Project Summary.** The Elliptic Curve Cryptography (ECC) inherently integrated within Ethereum does not align with the prevalent authentication frameworks utilized in the Web2 environment, making it difficult for users who are coming from the Web2 space to onboard. This discrepancy arises because Ethereum's ECC is based on secp256k1 curve, while the general Web2 infrastructures primarily utilize secp256r1 curve, also known as P256 curve.

Although it is technically feasible to develop a custom ECC primitive as a smart contract, this approach incurs significant gas costs, making it economically unfeasible for widespread adoption. To overcome this limitation, multiple Ethereum Improvement Proposals (EIPs) have been suggested: EIP-665<sup>1</sup>, EIP-7212<sup>2</sup>, RIP-7696<sup>3</sup>. The adoption of these EIPs would incorporate secp256r1-related functions directly as EVM opcodes, significantly reducing the cost for end users. This approach would facilitate a more cost-effective integration, enabling the widespread transition of users from the Web2 space to the Ethereum platform.

The Smooth Crypto Library (SCL) offers implementations of various EIPs that are optimized for gas efficiency.

Specifically, SCL consists of the following modules:

- ▶ libSCL\_ecdsab4.sol : implements ECDSA signature verification using the optimized DSM operator with a precomputed point.
- ▶ libSCL\_RIP665.sol: implements functions for the EdDSA signature verification.
- ▶ libSCL\_RIP7212.sol: Implements the new opcode for ECDSA P256 curve ECC signature verification.
- ▶ libSCL\_RIP7696\_2.sol : Implements the new opcode for a DSM operator that takes the precomputed points to be passed among other parameters.

The most challenging operation within the implementation of those functions is the Double Scalar Multiplication (DSM) operator. DSM is a central and frequently used operator for ECC. For any two points P, Q on a curve and two scalars u, v, this operator computes the coordinates of the point uP + vQ.

Veridise Audit Report: Smooth Crypto Library

<sup>1</sup> https://eips.ethereum.org/EIPS/eip-665

<sup>2</sup> https://eips.ethereum.org/EIPS/eip-7212

<sup>3</sup> https://ethereum-magicians.org/t/rip-7696-generic-double-scalar-multiplication-dsm-for-all-curves/ 19798

In SCL, for performance reasons, this operator is implemented using the Yul assembly language. This operator has been implemented in various forms, depending on the context in which it is used.

- ► SCL\_mulmuladdX\_fullgenW.sol: General DSM implementation.
- ► SCL\_mulmuladdX\_fullgen\_b4.sol : Optimized DSM implementation that takes extra parameters computed on the client side.
- ▶ SCL\_mulmuladd\_spec\_windowed.sol: Specific DSM implementation where the second point is taken to be the curve generator point.

**Code Review.** The project developers provided the source code of the Smooth Crypto Library contracts for review. The source code is original, written by Renaud Dubois from Smoo.th. The source code contains some documentation in a form of the README file and comments on functions in the source code. Also, the code documents the sub-expressions computed while implementing the mathematical formulae in inline comments. To facilitate the Veridise security analysts understanding of the code, the Smooth Crypto Library developer gave an overview of the most complex parts of the project and regularly answered specific questions via Telegram.

The source code contained several test suites, including test suites with integration tests covering ECDSA and EdDSA implementation, as well as WycheProof tests for ECDSA. Additionally, there were tests utilizing fuzzing techniques, implemented with the Foundry testing framework's fuzzing capabilities.

**Summary of Issues Detected.** The security assessment uncovered 6 issues, 2 of which is assessed to be of high severity. Specifically, V-SCL-VUL-001 describes an error in the opcode fallback implementation leading to potentially incorrect DSM computation. V-SCL-VUL-002 points out that signature verification is vulnerable to signature malleability attacks. The Veridise security analysts also identified 1 low-severity issue, and 3 informational findings.

**Recommendations.** After conducting the security review of the Smooth Crypto Library, the security analysts had a few suggestions for further improvements.

*Formal Verification.* Due to the fact that the program code is written in the low-level Yul language and includes many manual optimization techniques, and considering the importance of the cryptographic routines, it is recommended that at least the most critical parts of code, such as the DSM implementations, be subjected to formal verification.

*Comments.* The source code contains outdated or confusing comments that make it difficult to understand the logic behind the code. It would be helpful to update and organize the comments so that they are more clear and helpful for other developers.

*Test suite*. The project already has a fairly extensive test suite. However, we recommend expanding the test suite to include EdDSA tests with Wycheproof vectors and adding separate tests for the SHA512 hash function.

**Disclaimer.** We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

**Table 2.1:** Application Summary.

Name	Version	Type	Platform
Smooth Crypto Library	c559657	Solidity	Ethereum

Table 2.2: Current Status of All Identified Issues.

ID	Description	Status
V-SCL-VUL-001	Incorrect fallback implementation for RIP7696	Fixed at f40942c2
V-SCL-VUL-002	EdDSA verification is vulnerable to o malleability attacks	Fixed at 0af3c3ca
V-SCL-VUL-003	Function takes secret key as argument	Fixed at 2145023c
V-SCL-VUL-004	Maintainability observations	Fixed at 8eb4b152
V-SCL-VUL-005	Unspecified revert messages	Fixed at 83c25234
V-SCL-VUL-006	Gas optimizations	Fixed at 05d4a721

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	2	2	2
Medium-Severity Issues	0	0	0
Low-Severity Issues	1	1	1
Warning-Severity Issues	0	0	0
Informational-Severity Issues	3	3	3
TOTAL	6	6	6

Table 2.4: Category Breakdown.

Name	Number
Logic Error	1
Cryptographic Vulnerability	1
Information Leakage	1
Maintainability	1
Missing/Incorrect Events	1
Gas Optimization	1



### 3.1 Audit Goals

The engagement was scoped to provide a security assessment of Smooth Crypto Library's smart contracts. In our audit, we sought to answer questions such as:

- ▶ Is there any Yul-specific code defects, such as incorrect manual memory management or unbounded arithmetic overflow or underflow potential?
- ▶ Do the proposed implementations correctly reflect the Ethereum Proposals they are addressing?
- ► Are the DSM operators implemented correctly, according to the explanation given in the canonical texts?
- ▶ Are algorithms for a signature verification implemented correctly?
- ▶ Does the implementations handle all corner cases specific to ECDSA and EdDSA correctly?
- ▶ Is the SHA512 hash function implemented correctly?

## 3.2 Security Assessment Methodology & Scope

**Security Assessment Methodology.** To address the questions above, our security assessment process involved a combination of human experts-analysts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the *static analysis* technique: to identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy, uninitialized variables and unused code.

**Scope.** The scope of this security assessment is limited to the scl/src and scl/external folders of the source code provided by the Smooth Crypto Library developers, which contains the smart contract implementation of the Smooth Crypto Library, as well as SHA512 hash function implementation.

The following file has been removed out of the scope, as requested by the Smooth Crypto Library developers:

scl/src/lib/libSCL\_eccUtils.sol

**Methodology.** Veridise security analysts reviewed the reports of previous audits for Smooth Crypto Library, inspected the provided tests, and read the Smooth Crypto Library documentation. They then started a process of code review assisted by the static analyzers and testing tools. During the security assessment, the Veridise security analysts regularly interacted with the Smooth Crypto Library developers to ask questions about the code.

## 3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our security analysts weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR -
	Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
	Affects a large number of people and can be fixed by the user
Bad	- OR -
	Affects a very small number of people and requires aid to fix
Affects a large number of people and requires aid to fix	
Very Bad	- OR -
	Disrupts the intended behavior of the protocol for a small group of
users through no fault of their own	
Protocol Breaking Disrupts the intended behavior of the protocol for a large group	
users through no fault of their own	

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SCL-VUL-001	Incorrect fallback implementation for RIP7696	High	Fixed
V-SCL-VUL-002	EdDSA verification is vulnerable to	High	Fixed
V-SCL-VUL-003	Function takes secret key as argument	Low	Fixed
V-SCL-VUL-004	Maintainability observations	Info	Fixed
V-SCL-VUL-005	Unspecified revert messages	Info	Fixed
V-SCL-VUL-006	Gas optimizations	Info	Fixed

## 4.1 Detailed Description of Issues

### 4.1.1 V-SCL-VUL-001: Incorrect fallback implementation for RIP7696

Severity	High	Commit	c559657
Type	Logic Error	Status	Fixed
File(s)	libSCL_RIP7696_2.sol		
Location(s)	_fallback		
Confirmed Fix At	83c2523		

The libSCL\_RIP7696\_2.sol module contains an optimized implementation of the Double Scalar Multiplication operator, which is used in ECC. This optimization relies on user-provided extra parameters that make on-chain computations more gas-efficient.

The main computing operation that this module relies on is called ecGenMulmuladdX\_store . It expects the following parameters to be passed in the parameter array Q:

Qx, Qy, Qx2pow128, Qy2pow128, p, a, Gx, Gy, Gx2p128, Gy2pow128

The issue here is that the fields of the parameter Q are not set correctly. The initialization for the parameter Q[3] is missing, so it defaults to Q[3].

```
/* default is RIPB4 precompile as described in rip-b4 (name to be changed after
1
       submission)*/
        /* expected RIP data is: p, a, b, gx, gy, gx128, gy128, qx128, qy128*/
2
       function _fallback(bytes calldata input) internal view returns (bytes memory ret)
3
           if ((input.length != 384) ) {
4
5
               return abi.encodePacked(uint256(0));
6
           uint256 [10] memory Q;
8
           Q[4] = uint256(bytes32(input[0:32]));//p
           Q[5] = uint256(bytes32(input[32:64]));//a
10
           uint256 b = uint256(bytes32(input[64:96]));//b
11
12
           Q[6] = uint256(bytes32(input[96:128]));//x
13
           // ... skipped
14
           return abi.encodePacked(ecGenMulmuladdX_store(Q, u, v));
15
       }
```

**Snippet 4.1:** Snippet from \_fallback()

**Impact** The operation will produce incorrect results.

#### **Recommendation** It is recommended to:

- ► Clarify the input fields content and order. At the moment, the comment at the top of the function specifies it incorrectly. For example, it does not mention the coordinates of the second point.
- ► Set fields of Q according to the chosen input fields content and order.

#### 4.1.2 V-SCL-VUL-002: EdDSA verification is vulnerable to malleability attacks

Severity	High	Commit	c559657
Type	Cryptographic Vulnera	Status	Fixed
File(s)	libSCL_EIP665.sol		
Location(s)	Verify_LE, Verify		
Confirmed Fix At	83c2523		

Running the EIP665 implementation against the Wycheproof EdDSA tests revealed that the current implementation is vulnerable to the malleability attack. Due to the malleability of signatures, an attacker with access to a valid signature on the blockchain can generate other signatures that will pass verification.

**Impact** Some systems expect signatures to be unique. This will allow an attacker to perform tasks on behalf of a user.

**Recommendation** Apply the check mentioned in RFC8032 section 8.4 paragraph 2 to prevent signature malleability.

#### 4.1.3 V-SCL-VUL-003: Function takes secret key as argument

Severity	Low	Commit	c559657
Type	Information Leakage	Status	Fixed
File(s)	src/lib/libSCL_EIP665.sol		
Location(s)	SetKey()		
Confirmed Fix At	83c2523		

Function SetKey() takes a secret and returns the public key, the expanded private key and the signer secret.

Snippet 4.2: Snippet from SetKey() in libSCL\_EIP665.sol

The arguments and return values to and from public functions are visible on the chain.

**Impact** Users may accidentally use this function, revealing their signing secret.

**Recommendation** Remove this function or move it to testing utilities so this function does not get deployed on chain.

#### 4.1.4 V-SCL-VUL-004: Maintainability observations

Severity	Info	Commit	c559657
Type	Maintainability	Status	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	83c2523		

Following are few observations that make it difficult to maintain and update the codebase.

Repeated definition of constants

The Yul code uses areas of contagious memory areas for accepting arguments and for storing precomputed points at the beginning of the all of implementations of Double Scalar Multiplication operator, implemented in src/elliptic/
The following constants have duplicate declarations in the files mentioned above.

```
uint256 constant _Prec_T8=0x800;
uint256 constant _Ap=0x820;
uint256 constant _y2=0x840;
uint256 constant _zzz2=0x860;//temporary address for zzz2
uint256 constant _free=0x880;
```

**Snippet 4.3:** Definition of pre-computation areas and the address offsets of certain parameters.

▶ Use of literals

The function Red512Modq() reduces the the given 512-bit number to order of the curve.

**Snippet 4.4:** Snippet from Red512Modq()

This function uses literal that is defined in fields/SCL\_wei22519.sol

▶ Presence of dead code

The project contains several unused functions. The following functions were identified as unused throughout the project:

- In the file SCL\_Moduler.sol, the functions ModExp and pModInv are not used anywhere.
- In the file SCL\_mulmuladd\_spec\_windowed.sol, the function ecGenMulmuladdW is not used anywhere.
- In the file SCL\_sha512.sol, the function hash\_LE is not used anywhere.
- In the file libSCP\_EIP665.sol, the function edDecompressX is not used anywhere except tests.
- In the file Sha2Ext.sol, the function sha384 is not used anywhere.

• In the file LibBytes.sol, most of the functions are not used, except the following: slice and readbytes8

#### **Impact**

- ▶ Duplicate definitions and the use of literals can lead to errors that are difficult to detect. Additionally, any changes to constants would need to be propagated in all relevant locations.
- ▶ Unused functions increase the size of the codebase unnecessarily, making it more challenging to understand and maintain.

#### Recommendation

- ▶ Define the constants in a separate file and import it into the file where they are used.
- ▶ Use named constants instead of literals
- ▶ It is advisable to remove the unused functions, or otherwise justify their presence in comments.

#### 4.1.5 V-SCL-VUL-005: Unspecified revert messages

Severity	Info	Commit	c559657
Type	Missing/Incorrect Eve	Status	Fixed
File(s)	src/modular/SqrtMod_5mod8.sol		
Location(s)	SqrtMod		
Confirmed Fix At	83c2523		

The SqrtMod() function calculates the square root of a given value under the modular operation for the Ed25519 curve, if it exists. This function can revert in two locations as shown below.

1. The call to precompiled modexp contract fails

```
1 if iszero(
2
     call(
         \mathsf{not}(0), // amount of gas to send
3
4
         MODEXP_PRECOMPILE, // target
5
         0x00, // value in wei
         pointer, // argsOffset
6
7
         0xc0, // argsSize (6 * 32 bytes)
         _result, // retOffset (we override M to avoid paying for the memory
       expansion)
9
         0x20 // retSize (32 bytes)
10
11 ) { revert(0, 0) }
```

Snippet 4.5: Snippet from SqrtMod()

2. The square root does not exist

```
if(mulmod(result, result, p)!=self){
    revert();
}
```

**Snippet 4.6:** Snippet from SqrtMod()

The reverts here are without any value.

**Impact** Reverts without specific codes may confuse users and will be difficult for the users of this library to debug.

**Recommendation** Define constants for revert types and revert using those constants.

#### 4.1.6 V-SCL-VUL-006: Gas optimizations

Severity	Info	Commit	c559657
Type	Gas Optimization	Status	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	83c2523		

In the following locations, the security analysts identified redundant operations that cause excess gas expenditure.

- ▶ src/elliptic/SCL\_mulmuladdX\_fullgen\_b4.sol:275
  - The expression evaluated on line 275 is evaluated again on line 276 in the variable T4. A simple ordering of these statements allows us to reduce gas consumption
  - Leads to saving of approximately 1428000 gas on test testbench\_ecmulmuladd\_wei25519
- src/elliptic/SCL\_mulmuladdX\_fullgen\_b4.sol:173
  - The value of \_p is loaded in the beginning of the loop body defined on line 173
  - The value of \_p does not change in the loop body making the load of value of \_p redundant in every iteration.
  - Moving the load operation just before the loop reduces the gas consumption
  - On test testbench\_ecmulmuladd\_wei25519, this optimization shows gas saving of approximately 2817000

**Impact** Since these gas costs savings are in primitives, and since primitives are used frequently, the benefits of optimizations multiply in the callers of these primitives.

**Recommendation** Perform the suggested optimizations.